Extending Temporal Action Logic

Joakim Gustafsson
Department of Computer
and Information Science
Linköpings Universitet
S-58183 Linköping, Sweden
joagu@ida.liu.se

April 18, 2001

Abstract

An autonomous agent operating in a dynamical environment must be able to perform several "intelligent" tasks, such as learning about the environment, planning its actions and reasoning about the effects of the chosen actions. For this purpose, it is vital that the agent has a coherent, expressive, and well understood means of representing its knowledge about the world.

Traditionally, all knowledge about the dynamics of the modeled world has been represented in complex and detailed action descriptions. The first contribution of this thesis is the introduction of domain constraints in TAL, allowing a more modular representation of certain kinds of knowledge.

The second contribution is a systematic method of modeling different types of conflict handling that can arise in the context of concurrent actions. A new type of fluent, called influence, is introduced as a carrier from cause to actual effect. Domain constraints govern how influences interact with ordinary fluents. Conflicts can be modeled in a number of different ways depending on the nature of the interaction.

A fundamental property of many dynamical systems is that the effects of actions can occur with some delay. We discuss how delayed effects can be modeled in TAL using the mechanisms previously used for concurrent actions, and consider a range of possible interactions between the delayed effects of an action and later occurring actions.

In order to model larger and more complex domains, a sound modeling methodology is essential. We demonstrate how many ideas from the object-oriented paradigm can be used when reasoning about action and change. These ideas are used both to construct a framework for high level control objects and to illustrate how complex domains can be modeled in an elaboration tolerant manner.

Acknowledgements

The following people have contributed to this thesis with support, help, comments, or patience:
First of all I would like to express my deeply felt gratitude towards Patrick Doherty for his guidance and encouragement. Also many thanks to Jonas Kvarnström and Lars Karlsson for doing the tedious task of reading the manuscript numerous times. I am grateful to all members of KPLAB for much help and inspiration. I would also like to thank technical and administrative staff at IDA for support and assistance.

A cryptical thanks goes to the runners for interesting, if not fruitful discussions. Further I would like to thank my parents and sisters. Saving the most important for last, I would like to thank my beloved wife Henrietta Gustafsson for patience, support and love.

This work has been supported by the Swedish Research Council for Engineering Sciences (TFR) and the Wallenberg foundation

JOAKIM GUSTAFSSON, LINKÖPING, MAY 2001

Contents

Chapter 1

Introduction

Developing techniques for reasoning about action and change (RAC) has been an important topic of research in artificial intelligence from its inception. This is natural, because the ability to foresee the results of actions is something that distinguishes intelligent behavior. In particular it is the basis for forming plans or predicting the result of the execution of a plan.

The approach taken in this thesis follows the tradition of using logic to define properties of a world and the actions available to an agent. The world history is represented by a sequence of states. Each state is a snapshot of the world at a particular time-point. A state is an assignment of values to variables called *fluents*. The idea is that the user formulates how actions influence the world either directly in logic or in a surface language that is translated into logic via a macro-translation. The result is combined with a logic theory consisting of axioms describing how states relate to each other and other technical details. The possible sequences of states can then be inferred from the combined theory via an inference mechanism.

The slow development of this area is hardly surprising because the goal is quite general and the solution quite complex. For example, we want to model not only simple sequences of deterministic single step actions but also the following:

- **Context dependency.** The result of an action may depend on the state in which it is invoked.
- **Nondeterminism.** For a given state, the execution of a nondeterministic action may lead to several different possible resulting states.
- **Ramification.** Any action can have indirect effects, possibly long chains of dependencies. If all of these side-effects have to be expressed

directly in the actions, the action descriptions can become very large and complex.

- **Concurrency.** Actions can overlap in time. Several different types of conflicts can result from this.

In order to reason about actions, an explicit or implicit model of time is needed. For example, an action may be invoked at a certain time-point, and may be executed during an interval of time or possibly instantaneously at a single point in time. The two most popular ways to represent time are either to let time progress only when an action is executed (these approaches are called event-based), or to have an explicit timeline and specify the time-points when each action starts and stops.

Time progression is the root of the frame problem. Any single action normally changes only a few of the many fluents that describes a complex world. When using logic, we need a concise way of ensuring that fluents not explicitly affected by a certain action invocation retain their values from before the action was invoked, without having to state this explicitly for each unaffected fluent. This problem is arguably solved today for particular well-defined classes of problems.

Unfortunately, the existing solutions are forced to use some kind of non-monotonic construction. The choice has often been circumscription [93, 60] or its semantical counterpart, preferential entailment [96, 90]. Due to the second-order nature of circumscription, it is hard to design efficient algorithms for reasoning in such logics. It is a great advantage if we can remain in first-order logic, or use restricted forms of second-order logic theories that can be reduced to logically equivalent first-order logic theories. All logics developed in this thesis can be reduced in this manner. It is even questionable whether first-order theories meet the requirements of representational efficiency but there are many results that can be used in this case.

1.1 Motivation and Topics

The intention of this thesis is to investigate solutions to the following problems in the research area of reasoning about action and change.

The ramification problem: Since reasonable solutions to the frame problem have been found, much of the attention in the research community has now shifted to the problem of modeling side-effects of actions. The classical

solutions to the frame problem consider actions to be the only way to affect the values of fluents, but it is infeasible to provide an exhaustive action description for each possible state of the world. This is called the *ramification problem*, a name originally suggested by Ginsberg and Smith [39] and inspired by Finger [31]. Ginsberg and Smith provide the following description:

> The difficulty is that it is unreasonable to explicitly record all of the consequences of an action, even the immediate ones. [...] For any given action there are essentially an infinite number of possible consequences that depend upon the details of the situation in which the action occurs.

Action descriptions should therefore be restricted to the part of the world that are directly affected by the action. The indirect effects should be handled by some type of *dependency constraints* that describe the behavior of the world in terms of dependencies among fluents.

The ramification problem is one of the main topics of this thesis. In Chapter 2 we present PMON, a logic that handles some aspects of the problem using dependency laws. This solution handles a broad class of domains, but the ramification problem is still far from solved. For example, like many other approaches, the PMON approach has some trouble dealing with cycles of causal dependencies. We also present two different ways of eradicating this limitation.

The dependency laws developed for ramification in Chapter 2 have proven extremely versatile and are used in the remainder of the thesis, with minor modifications, for modeling and solving additional problems.

Concurrency: When action invocation intervals are permitted to overlap, or when delayed effects of actions are allowed, a fluent might be influenced in several different ways at the same time. This might lead to contradictions or other unintuitive results. Many approaches to this problem use only instantaneous actions and handle conflicts by explicitly, in the action definitions, describing what happens when several actions are invoked concurrently. We show some weaknesses with this approach and suggest an extension to the logic we constructed to handle ramifications. This extension, described in Chapter 5, deals with conflicts and many other problems associated with concurrency.

Elaboration tolerance: McCarthy [75] defines elaboration tolerance as follows:

> A formalism is *elaboration tolerant* to the extent that it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances.

The importance of elaboration tolerance increases as the worlds we model grow larger and more complex and as we attempt to model them in increasingly greater detail. If we aim to model more realistic scenarios than simple toy examples there is a need to develop mechanisms for incrementally constructing them in a structured and principled manner. The object-oriented paradigm has been developed for these purposes with respect to ordinary programming languages. In Chapter 7 we show how the ideas of an object-oriented approach to scenario construction can be applied to our logic. In Chapter 9 we apply the object-oriented methodology to some larger examples. One of the examples presented in that chapter is the cannibals and missionaries problem, with the elaborations originally described by McCarthy.

1.2 Organization

This thesis is based on work done in co-operation with my thesis advisor Patrick Doherty, Lars Karlsson and Jonas Kvarnström.

Chapter 2 presents the state of the PMON logic before the modifications presented in this thesis.

Chapter 3 contains a selected survey of other approaches used in solving the ramification problem.

Chapter 4 is based on a paper presented at KR'96, written in co-operation with Patrick Doherty [42]. This chapter shows how PMON can be extended to deal with indirect effects of actions using dependency laws. Further, it discusses the problems associated with causal cycles and proposes two solutions. The second of these is based on an unpublished paper written in co-operation with Jonas Kvarnström [43].

Chapter 5 presents TAL-C [49, 47], which is a modification of PMON designed to handle concurrency and is based on the ideas from Chapter 4. The formal definition of TAL-C can be found in Appendix C.

Chapter 6 explores how one can model delayed effects of actions in TAL. Conflicts due to delayed effects of actions have much in common with conflicts caused by concurrency. Furthermore, the explicit timeline combined with the

ability to use separate time-points in the precondition and effect parts of a dependency law make our logics well suited for handling delayed effects of actions. Modeling delayed effects of actions is difficult for many of the other approaches proposed for reasoning about action and change. This chapter is based on a paper by Lars Karlsson, Joakim Gustafsson and Patrick Doherty published at ECAI'98 [49].

Chapter 7 provides a set of tools for structuring large domains using object-oriented concepts. During the work on PMON and TAL it become evident that there is a lack of a methodology of constructing modular, elaboration tolerant domains and the object-oriented paradigm is a well known solution to these problems. The work on object-orientation is based on a paper by Joakim Gustafsson published at SCAI'01 [41].

Chapter 8 describes how the object-oriented methodology can be used to model objects that behave as controllers, automatically invoking actions at suitable points in time. This is in contrast to classical TAL scenarios, where the user specifies a finite list of timed action instances to be executed.

Chapter 9 provides three slightly larger scenarios where all of the tools developed in the thesis are used. The scenario descriptions are found in Appendices D, E and F.

Chapter 10 presents some of the conclusions we can draw from the thesis and discusses future work.

1.3 History of PMON and TAL

The logical formalisms used in this thesis have gone through a number of iterations with various extensions which have resulted in the use of different naming conventions in the published papers. In this section, we provide a short history of the development of these formalisms to provide a context for the remainder of the thesis.

We begin the history of the logic now known as TAL (Temporal Action Logic) with the Features and Fluents framework [90] developed by Sandewall. Features and Fluents is a systematic approach to the representation of knowledge about dynamic systems that includes a framework for assessing the range of applicability of existing and new logics of action and change. Several logics of action and change are introduced and assessed as being correct for particular classes of action scenario descriptions. The scenario class $\mathcal{K}$-IA and one of its associated entailment relations, PMON (*Pointwise Minimization of Occlusion with Nochange premises*), requires explicit, correct and accurate knowledge, and deals with nondeterministic actions, incomplete specification of state and the timing of actions, and observations at arbitrary

time-points. Doherty [19, 20] provides a syntactic characterization of PMON in terms of classical logic and circumscription and shows that for the $\mathcal{K}$-IA class, the circumscription axiom can be reduced to a first-order formula. Doherty's PMON logic provides a surface language together with a translation function from the surface language to classical logic.

Note that although PMON is an entailment relation in Features and Fluents, the original logic developed by Doherty, based on PMON, was also called PMON, but the emphasis was on syntactic categorization of the entailment method.

Although the logic PMON is assessed correct for a broad class of action scenarios, it is restricted to actions that do not permit indirect effects, and actions cannot occur concurrently. It deals with the frame problem in isolation but not with ramification. The further development of the PMON logic follows the graph in Figure 1.1. Inspired by the frame construct in Kartha and Lifschitz [50], Doherty and Peppas [27] extended PMON with the frame construct to deal with various types of ramification. The resulting logic was called PMON(R). The frame construct is a way to globally specify a set of fluents that are in the frame and subject to inertia, while all other fluents may vary freely. This approach has several weaknesses, among others the inability to handle chains of dependencies. For this reason, work continued with developing a modification able to handle a more general class of scenarios than PMON(R). This led to the development of PMON(RCs) by Gustafsson and Doherty [42] described in Chapter 4. PMON(RCs) is a modification of PMON where a macro handling directional dependency laws is added to the surface language. PMON(RCs) also uses first-order predicate logic as compared to PMON which uses propositional logic. PMON(RCs) was modified and extended into PMON$^+$ in Doherty [21], where the formal proofs underlying the logic were gathered and presented. At this stage the family of PMON logics was renamed "TAL" (Temporal Action Logics) to avoid confusion with the entailment relation.

PMON$^+$ has formed a basis from which two modified versions emerged. The first extension is TAL-C by Karlsson and Gustafsson [47], who investigate how the logic can be extended to model concurrency. We describe this logic in Chapter 5 and provide a formal specification in Appendix C. The second extension is TAL(seq) which applies ideas from Thielscher [98] to TAL. Section 4.5 describes of TAL(seq). The latest version of the logic, TAL 2.0 by Doherty et al. [22], is basically the same logic as TAL-C, but with some minor modifications. It is intended to provide a uniform presentation of the current stable kernel of the TAL family of logics.

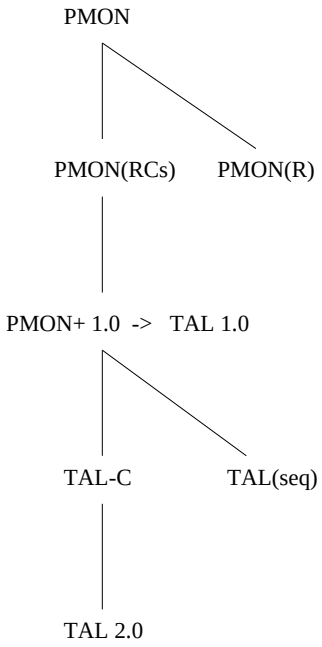

Figure 1.1: The evolution from PMON onwards.

1.4 Ramification Terminology

Some terminology regarding ramifications are necessary to clarify before we move on.

In Chapter 4 we introduce constraints of the form $\alpha \gg \beta$, called causal rules or causal constraints, loosely interpreted as "α causes β". We will call α the trigger and β the effect. There is sometimes also a precondition γ which means that the rule has the form $\gamma \rightarrow \alpha \gg \beta$. These forms provide a means of representing some limited forms of causal laws or rules.

In Chapter 5, we develop the logic TAL-C and introduce a more general means of representing causal rules using three new operators R, I and X. We have chosen to call the more general rules dependency rules or dependency constraints instead of causal rules. This is to avoid many of the associations the word "causality" carries. The operators provide a generic tool to express dependencies between fluents and can be used to model both *causal dependency laws* and *explanatory dependency laws*. Causal dependency laws are of the kind "A causes B", for example "turning the switch causes the lamp to go on". Explanatory dependency laws work in the opposite direction, "If

B is true, the explanation is that A is true", for example "if the lamp is on, the explanation must be that the switch has been turned on". Causal and explanational dependency laws will be handled in the same way since the difference is conceptual rather than technical.

The next chapter presents a short description of the state of the PMON/TAL logic before the work presented in this thesis began [19, 20].

Chapter 2

Basic PMON

Many reasoning problems involving action and change can be conveniently represented in terms of *(action) scenario descriptions*. Scenario descriptions can be described as partial specifications of initial and other states of a system, combined with descriptions of the actions that have occurred together with their timing. The "Yale Shooting Scenario" [44] and "Stanford Murder Mystery" [8] problems are well known examples of scenarios. Scenario descriptions can be formalized directly in terms of a logic language, or, for convenience, described in a higher level macro language which is then compiled into a logical language. In our framework, we will represent action scenarios in a narrative language $\mathcal{L}$(ND), which is translated into a standard logical language $\mathcal{L}$(FL). All formal reasoning will be done using $\mathcal{L}$(FL) together with appropriate circumscription policies for modeling inertia. Detailed descriptions of both languages and the translation process may be found in Doherty [19, 20] and in Appendices A and B. Please refer to the book Features and Fluents [90] by Erik Sandewall for an interesting and detailed discussion of the philosophy behind modeling action and change using this approach.

The description in this chapter loosely follows previous work [25, 19, 20, 42] but contains many simplifications for purposes of readability, such as only dealing with boolean domains and the absence of a more precise definition of fluents. It will, however, provide sufficient detail to provide a basic understanding of the original PMON languages.

2.1 The Language $\mathcal{L}$(ND)

The formal syntax for specifying scenario descriptions is defined in terms of the surface language $\mathcal{L}$(ND), consisting of action occurrence statements,

action law schemas, and observation statements, labeled with the symbols occ, acs, and obs, respectively.

Example 2.1.1
The well known Stanford Murder Mystery scenario is shown below using the $\mathcal{L}$(ND) syntax:

Scenario Description 2.1
obs_1 $\mathtt{t}_0 = 0 \wedge \mathtt{t}_1 = 10$
obs_2 $[\mathtt{t}_0]\mathsf{alive}$
obs_3 $[\mathtt{t}_1]\neg\mathsf{alive}$
occ_1 $[2,6]\mathsf{Fire}$
acs_1 $[t_1,t_2]\mathsf{Fire} \rightsquigarrow [t_1]\mathsf{loaded} \rightarrow [t_1,t_2](\mathsf{alive} := F \wedge \mathsf{loaded} := F)$ □

Given a scenario description Υ, consisting of statements in the surface language $\mathcal{L}$(ND), these statements are translated into formulas in the many-sorted first-order language $\mathcal{L}$(FL) via a two-step process. In the first step, action schemas in Υ are instantiated with each action occurrence statement of the same name, resulting in what are called *schedule statements*, where each schedule statement is labeled with the symbol scd. The schedule statements replace the action schemas and action occurrence statements. The result of the first step is an *expanded (action) scenario description*, Υ', consisting of both schedule and observation statements.

Example 2.1.2
The expanded scenario description associated with Example 2.1.1 is shown below:

obs_1 $\mathtt{t}_0 = 0 \wedge \mathtt{t}_1 = 10$
obs_2 $[\mathtt{t}_0]\mathsf{alive}$
obs_3 $[\mathtt{t}_1]\neg\mathsf{alive}$
scd_1 $[2]\mathsf{loaded} \rightarrow [2,6](\mathsf{alive} := F \wedge \mathsf{loaded} := F)$ □

In the second step of the translation process, macro-translation definitions are used to translate statements in Υ' into formulas in $\mathcal{L}$(FL). Before translating the example, we must first define $\mathcal{L}$(FL).

2.2 The Language $\mathcal{L}$(FL)

The base language $\mathcal{L}$(FL) is a many-sorted first-order logic with equality. For the purposes of this chapter, we assume two sorts[1]: a sort $\mathcal{T}$ for time and a sort $\mathcal{F}$ for fluents. The language includes the predicate symbols *Holds* and *Occlude*, of type $\mathcal{T} \times \mathcal{F}$. The intuition behind $Holds(t, f)$ is that fluent f is true at time-point t, and $Occlude(t, f)$ can loosely be interpreted to mean that f is exempt from the inertia assumption at time-point t.

The numerals $0, 1, 2, \ldots$ and the symbols $\mathtt{t}_0, \mathtt{t}_1, \ldots$ will be used to denote constants of type $\mathcal{T}$ and the symbols $t_0, t_1, \ldots$ will be used to denote variables of type $\mathcal{T}$. We define the *set of temporal terms* to be the closure of the temporal variables and temporal constants of the language under the operators $+$ and $-$. A fluent is a function of time with the boolean truth values as range. The symbols $f_1, f_2, \ldots$ will be used to denote variables of type $\mathcal{F}$. We assume an appropriate set of function symbols of proper arity for fluent names (for example alive, at).

An *atomic formula* is defined as any formula of the form $Holds(t, f)$ or $Occlude(t, f)$, where t is a temporal term and f is a fluent. The set of formulas of $\mathcal{L}$(FL) is defined as the closure of the atomic formulas under the boolean connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) and the universal and existential quantifiers ($\forall$, $\exists$).

The intended interpretation for $\mathcal{T}$ is linear discrete time where $\mathcal{T}$ is considered isomorphic to the natural numbers. Since there is no axiomatization for time interpreted as the natural numbers, we either assume an interpreted language, settle for something less such as "integer-like, discrete flow of time with a first moment" which is axiomatizable [33], or assume a sound but incomplete axiomatization of the flow of time. In practice, we will be using a specialized temporal constraint module for reasoning about time which is normally sound, but incomplete.

2.3 From $\mathcal{L}$(ND) to $\mathcal{L}$(FL)

As stated in Section 2.1, the second step of the translation process uses macro-translation definitions to translate statements in Υ' into formulas in $\mathcal{L}$(FL). We need a few preliminary definitions which will prove useful both here and in the definition of causal constraints in a later chapter. A *elementary reassignment formula* is of the form $f := T$ or $f := F$, where f is a

[1]Note that we have restricted this chapter to only fluents with boolean value domains. Appendix A contains the complete definition, where this restriction is lifted.

fluent. A *fluent formula* is any boolean combination of fluents. An *elementary scenario formula* is of the form $[t]\gamma$, where t is a temporal term in $\mathcal{L}(\mathrm{FL})$ and γ is a fluent formula. A *scenario formula* is any boolean combination of elementary scenario formulas.

Let γ and δ denote fluent formulas and ϵ denote a fluent (possibly negated). Let C be any of the logical connectives $\wedge$, $\vee$, or $\rightarrow$.

Any elementary scenario formula can be reduced to a boolean combination of elementary scenario formulas of the form $[t]\epsilon$:

$$[t](\delta\ C\ \gamma) \stackrel{\text{def}}{=} [t]\delta\ C\ [t]\gamma.$$

The following list of macro-translation definitions should suffice to provide the general idea behind the translation from $\mathcal{L}(\mathrm{ND})$ to $\mathcal{L}(\mathrm{FL})$:

$$\begin{aligned}
[s,t]\delta &\stackrel{\text{def}}{=} \forall t_1.s \leq t_1 \leq t \rightarrow [t_1]\delta \\
[s,t)\delta &\stackrel{\text{def}}{=} \forall t_1.s \leq t_1 < t \rightarrow [t_1]\delta \\
[t]\epsilon &\stackrel{\text{def}}{=} \mathit{Holds}(t,\epsilon) \\
[s,t]\epsilon := T &\stackrel{\text{def}}{=} \mathit{Holds}(t,\epsilon) \wedge \forall t_1(s < t_1 \leq t \rightarrow \mathit{Occlude}(t_1,\epsilon)) \\
[s,t]\epsilon := F &\stackrel{\text{def}}{=} \mathit{Holds}(t,\neg\epsilon) \wedge \forall t_1(s < t_1 \leq t \rightarrow \mathit{Occlude}(t_1,\epsilon)) \\
\mathit{Holds}(t,\neg\epsilon) &\stackrel{\text{def}}{=} \neg\mathit{Holds}(t,\epsilon).
\end{aligned}$$

The translation of Υ' in Example 2.1.2 into $\mathcal{L}(\mathrm{FL})$ is shown below:

obs_1 $\mathtt{t}_0 = 0 \wedge \mathtt{t}_1 = 10$
obs_2 $\mathit{Holds}(\mathtt{t}_0, \mathsf{alive})$
obs_3 $\neg\mathit{Holds}(\mathtt{t}_1, \mathsf{alive})$
scd_1 $\mathit{Holds}(2, \mathsf{loaded}) \rightarrow$
$[\neg\mathit{Holds}(6, \mathsf{alive}) \wedge \neg\mathit{Holds}(6, \mathsf{loaded}) \wedge$
$\forall t(2 < t \leq 6 \rightarrow \mathit{Occlude}(t, \mathsf{alive})) \wedge$
$\forall t(2 < t \leq 6 \rightarrow \mathit{Occlude}(t, \mathsf{loaded}))]$

Note that although the labels are not part of the language $\mathcal{L}(\mathrm{FL})$, they are retained after translation. The labels are directly correlated with the partitioning of formulas used in the circumscription policy described in the next section. The notation

$$\Gamma_C = \Gamma_{\mathrm{OBS}} \cup \Gamma_{\mathrm{SCD}} \cup \Gamma_{\mathrm{UNA}} \cup \Gamma_{\mathrm{T}}$$

is used for a scenario description in $\mathcal{L}(\mathrm{FL})$, where Γ_{OBS} and Γ_{SCD} contain the observation and schedule statements in the scenario, Γ_{T} contains the axiomatization for the flow of time (when provided), and Γ_{UNA} contains the

appropriate unique name axioms for the sorts $\mathcal{T}$ and $\mathcal{F}$. In the rest of the thesis, we will use the convention of suppressing Γ_{UNA} and Γ_{T}, assuming they are provided with every theory. In addition, we will use the notation Γ_X, where X is an acronym such as OBS, for a finite set of formulas or their conjunction in contexts where this makes sense.

2.4 PMON Circumscription

In this section, we will describe the mechanisms that cause inertia in the fluents that are not influenced by any action. We will describe the intuition behind the use of occlusion, introduce a nochange axiom, describe the filtered minimization technique, provide a circumscription policy which uses occlusion, the nochange axiom, and filtering, and show that the second-order circumscription policy for any theory in the $\mathcal{K}$–*IA* class of action scenarios is reducible to a first-order theory.

2.4.1 Occlusion

The use of the occlusion concept and its representation in terms of the predicate *Occlude* has already been shown to be quite versatile in providing solutions to a number of open problems associated with the representation of action and change. Although occlusion is related to the use of an abnormality predicate together with an inertia assumption, there are some differences. The main difference is perspective. *Occlude* is used to provide a fine-grained means of excluding particular fluents at particular points in time from being subject to what are normally very strong inertia assumptions. In fact, in retrospect much of the progress made in solving a number of problems stemming from the original Yale Shooting Scenario has been the gradual relaxation of strict inertia in dealing with nondeterminism, postdiction and in the current case, indirect effects and delayed effects of actions. It is the fine-grained use of *Occlude* together with the filtered minimization technique, where *Occlude* is minimized in only parts of theories, that contributes to the simplicity of the solution. Filtering minimizes the need for complex minimization strategies. In fact, most of the time, the minimization policy involves little more than applying predicate completion to *Occlude* relative to part of a theory.

Recall the scenario description in Section 2.3. Associated with each action type in a scenario is a subset of fluents that are potentially influenced by the action (those fluents in the right-hand side of a rule). If the action has duration, then during its execution, it is not known in general what value the

influenced fluents have. Since the action performance can potentially change the value of these fluents at any time, all that can generally be asserted is that at the end of the duration the fluent is assigned a specific value (note that additional constraints can be provided for activity within the duration). To specify such behavior, the *Occlude* predicate is used in the definition of reassignment expressions which in turn are used as part of the definition of an action schema. In order to specify actions with durations and indeterminate effects of actions properly, it should be clear that fluents directly set by an action should be occluded during the execution of the action. In Lifschitz's [62] terminology, occluded fluents are simply frame-released fluents.

The predicate *Occlude* takes a time-point and a fluent as arguments. The definition for a reassignment expression $[t_1, t_2]\delta := T$ used in an action occurrence statement as follows:[2]

$$Holds(t_2, \delta) \wedge \forall t(t_1 < t \leq t_2 \rightarrow Occlude(t, \delta)).$$

Referring to our previous example, it can be observed that the occlusion specification is automatically generated by the translation process from $\mathcal{L}$(ND) to $\mathcal{L}$(FL). Occlusion specifies what fluents may change at what points in time. The nochange axiom described next specifies when a fluent is *not* permitted to change value.

2.4.2 The PMON Circumscription Policy

Let Γ_{NCG} denote the following nochange axiom,

$$\forall f, t(Holds(t, f) \oplus Holds(t+1, f) \rightarrow Occlude(t+1, f)), \qquad (2.2)$$

where the connective $\oplus$ denotes exclusive-or. The axiom states that if a fluent f is not occluded at $t+1$ then it can not change value from t to $t+1$. This axiom, together with the observation axioms, will be used to filter potential histories of action scenarios.

Filtered preferential entailment is a technique originally introduced by Sandewall [89] for dealing with postdiction. The technique is based on distinguishing between different types of formulas in a scenario description and applying minimization to only part of the scenario, or different minimization policies to different parts of the scenario. In this particular case, we will distinguish between schedule statements Γ_{SCD} and the rest of the scenario description. The idea is to minimize the *Occlude* predicate relative to the

[2]For $[t_1, t_2]\delta := F$, simply negate the *Holds* predicate.

schedule statements and then filter the result with the observation formulas $\Gamma_{\rm OBS}$ and the nochange axiom $\Gamma_{\rm NCG}$. The minimization policy generates potential histories where the possibility for change is minimized. The potential histories are then filtered with the observations, which must hold in any valid history, and with the nochange axiom which filters out any spurious change not explicitly axiomatized by the actions. More formally, instead of using the policy

$$\Gamma_{\rm NCG} \wedge \Gamma_{\rm C} \wedge \mathit{Circ}_{\rm SO}(\Gamma_{\rm C}(\mathit{Occlude}); \mathit{Occlude}),$$

where Circ_{SO} denotes standard second-order circumscription, PMON circumscription is defined using the policy

$$\Gamma_{\rm NCG} \wedge \Gamma_{\rm C} \wedge \mathit{Circ}_{\rm SO}(\Gamma_{\rm SCD}(\mathit{Occlude}); \mathit{Occlude}). \tag{2.3}$$

Observe that the circumscription policy is surprisingly simple, yet at the same time assessed correct for the broad ontological class $\mathcal{K}$–*IA*. One simply minimizes the *Occlude* (frame-released) predicate while leaving *Holds* fixed in that part of the theory containing the action occurrences and then filters the result with the nochange (inertia) axiom and the observation axioms.

Although $\mathit{Circ}_{\rm SO}(\Gamma_{\rm SCD}(\mathit{Occlude}); \mathit{Occlude})$ is a second-order formula, it can be shown using two results by Lifschitz [61] and the fact that *Occlude*-atoms only occur positively in $\Gamma_{\rm SCD}$, or through the use of predicate completion, that it is reducible to an equivalent first-order theory. Details may be found in Doherty [20, 19].

Example 2.4.1 (Cont. example 2.1.2)
The final $\mathcal{L}$(FL) theory of example 2.1.2, excluding $\Gamma_{\rm UNA}$, $\Gamma_{\rm NCG}$ and $\Gamma_{\rm T}$ is as follows:

$$\begin{aligned}
&\mathtt{t}_0 = 0 \wedge \mathtt{t}_1 = 10 \wedge \\
&\mathit{Holds}(\mathtt{t}_0, \mathsf{alive}) \wedge \\
&\neg \mathit{Holds}(\mathtt{t}_1, \mathsf{alive}) \wedge \\
&\mathit{Holds}(2, \mathsf{loaded}) \rightarrow \\
&\qquad \neg \mathit{Holds}(6, \mathsf{alive}) \wedge \neg \mathit{Holds}(6, \mathsf{loaded}) \wedge \\
&\forall t, f.\ \mathit{Occlude}(t, f) \leftrightarrow \\
&\qquad [\mathit{Holds}(2, \mathsf{loaded}) \wedge 2 < t \leq 6 \wedge \\
&\qquad (f = \mathsf{alive} \vee f = \mathsf{loaded})]
\end{aligned}$$

It entails that the gun must have been loaded at the initial time-point, which is the correct result. □

In the following chapters we will examine some of the most well-known approaches to ramification and then present a solution within the PMON framework that continues to satisfy the requirement that the use of circumscription is always reducible to a logically equivalent first-order theory.

Chapter 3

Survey of Ramification Approaches

This chapter contains an overview of some of the most well-known approaches to ramification. Although some of these approaches also cover solutions to other problems, such as the frame problem, the main subject of this chapter is how they handle the ramification problem. The terminology has been left unchanged as far as possible; footnotes are provided for comparison with our terminology. We will place more emphasis on the so-called causal approaches, and only briefly sketch the others. Sandewall [91] provides a comparative assessment of both causal and non-causal approaches.

In Section 3.1, we identify some important properties of approaches to ramification which we will refer to in the discussions of the different approaches. Next, Section 3.2 briefly describes some non-causal approaches. Section 3.3 describes some of the more important causal approaches in more detail. Finally, Section 3.4 contains an overview of the different approaches considered in this chapter.

3.1 Important Aspects of Ramification

The three main factors that will be investigated in the different formalisms discussed in this chapter are expressivity, the ability to handle theories with causal cycles, and the representation of time.

Expressivity Although all of the approaches represent causal rules as logical formulas, the expressivity of these formulas varies between the different approaches. For example, some approaches use propositional logic while others use predicate logic. Other differences relate to whether preconditions are allowed and how general the effect part of a rule can be.

Some approaches allow the left-hand side of a dependency law to contain both triggers and preconditions. The difference between a precondition and a trigger is that the precondition is used to detect values of fluents while triggers are used to detect change of values in fluents. As we will see, not all formalisms allow preconditions to dependency laws.

The formalisms also differ in how general the postcondition of a dependency law can be. Some approaches allow arbitrary first-order formulas, while others limit the postcondition to a literal.

Causal cycles Causal cycles can be difficult to handle in logic. Therefore some approaches require stratified theories where no casual cycles are present. A full discussion on the problems associated with causal cycles is found in Section 4.4.

Representation of time The representation of time fills an important role when we want to represent delayed effects and actions with durations. Some of the approaches use an explicit timeline while others are event-based.

3.2 Non-Causal Approaches

Non-causal approaches, which do not support explicit causal directionality, will only be briefly mentioned.

Ginsberg and Smith The first to discuss ramifications, in the area of reasoning about action and change, were Ginsberg and Smith [38]. They use a possible worlds approach which works in a STRIPS-like manner [30]. Only a single model of the world is maintained, which is updated to reflect the results of any particular action. However, a complete description of the consequences of actions is not required. The model of the world is updated inferentially from actions and domain constraints. The domain constraints specify the relationships between different facts in the world. Inference here means trying to find the nearest possible world to the current one based on a standard notion of "nearness".

The stuffy room problem is presented and discussed. It illustrates the fact that straightforward minimization of change can give unintuitive results. Basically one wants to specify that a room is stuffy if both ducts leading air into the room get blocked. Now suppose that one duct is blocked and an action blocking the other duct is executed. The result is two models, one where the room gets stuffy, and one unintuitive model where the other duct becomes unblocked, and the room remains not stuffy. We cannot select one of these models over the other since neither of them is a subset of the other. Ginsberg and Smith consider both models legal. The authors argue that since this is a result of the knowledge we have put into the system, this is what we should expect. More knowledge has to be added in order to remove the unwanted model.

Kartha and Lifschitz As an extension to the $\mathcal{A}$ language [35], Kartha and Lifschitz [50] present $\mathcal{AR}_0$. The basic idea is to divide the fluents into frame and nonframe fluents, where only the frame fluents are subject to inertia. The nonframe fluents should be completely determined by the domain constraints. The frame fluents can of course be temporarily released from the persistence assumption if they are caused to change by actions. The surface language is translated into a nested abnormality theory and minimized with respect to abnormality.

Doherty and Peppas PMON(R) by Doherty and Peppas [27] is a version of the PMON logic inspired by the work of Kartha and Lifschitz, but with a time argument to the frame predicate to make it more fine-grained. In PMON, inertia is enforced using a nochange axiom; the key here is to apply the nochange axiom only to the frame fluents, thereby excluding the non-frame fluents from the persistence assumption.

Lin and Reiter Lin and Reiter [69] extend the situation calculus to deal with some aspects of the ramification problem. This is done with a specialized (and technically quite complicated) priority minimization policy. They also discuss the interpretation of domain constraints as qualification of actions, which is a problem pointed out by Ginsberg and Smith [38].

3.3 Causal Approaches

The basic idea behind the causal approach to dealing with ramification is to explicitly model the direction of dependencies between fluents. This dependency information can be incorporated as domain constraints or provided as a relation between fluents. For historical reasons we will refer to these kinds of methods as causal approaches, even though the word "causal" is not well chosen. The types of dependencies expressible by these methods are not limited to dependencies that are strictly causal. McCain and Turner [72] suggest the following two sentences to distinguish between two different kinds of knowledge, where Ψ and Φ are sentences.

- The fact that Ψ causes the fact that Φ.
- Necessarily, if Ψ then the fact that Φ is caused. [1]

We will use the term "dependency law" to denote both these types of laws.

Example 3.3.1
If a box is pushed, then it is caused to move. But we might be more interested in finding explanations instead, so the rule we want to formulate might be that the box is stationary if it is not pushed. Using a "causal" approach there is nothing that prevents us from writing rules of the kind $\neg moves \gg \neg pushed$, even though $\neg pushed$ is not caused by $\neg moves$. What we have is a dependency relation between the box and the forces acting upon it. If the box is stationary, we can determine that it is not pushed. In the words of McCain and Turner's second sentence from above: "Necessarily, if the box is stationary then the fact that it is not pushed is caused". The reason for the box not being pushed needs not be explicitly stated in this sentence. □

3.3.1 Early Work

Causality between events is mentioned by Georgeff [37]. He distinguishes between the case where an event causes the simultaneous occurrence of another event, and the case in which an event causes the occurrence of a consecutive event. A predicate Causes is used in both cases to represent the causal relationship between events. $\mathsf{Causes}(\Phi, e_1, e_2)$ means that if Φ is true and the event e_1 happens, then so does event e_2.

Pearl was one of the first to argue for a primitive notion of causality ([80] and [81]). Since his method is probabilistic, it is difficult to compare with

[1] Note that Ψ does not have to be the cause of Φ.

the approaches focused upon in this thesis and we will not describe these alternatives here. Many of the ideas can be seen in other work inspired by the work of Pearl, for example Geffner [34], Lin [67], McCain and Turner [71], and Grünwald [40]. Especially Grünwald's approach seems to bring the ideas of Pearl closer to our research area.

Other early work on causality includes Lansky [56], McDermott [77], Shoham [95] and Allen [3].

Three influential causal methods were presented at IJCAI'95. Thielscher [98] suggests a fixpoint-oriented approach, McCain and Turner [71] present both a fixpoint and a declarative approach, and Lin [67] proposes an extension to the situation calculus. All these three are presented below and have in common that they support a means of explicitly specifying the directionality of dependencies between fluents.

3.3.2 McCain and Turner, 95

Terminology

The method presented by McCain and Turner [71] uses a standard language of propositional logic, based on a fixed set of atoms. An interpretation for the language is represented by a maximal consistent set of literals.

Background knowledge is given in the form of *state constraints* and *causal laws*[2]. A standard example of a state constraint is $\mathsf{Walking} \rightarrow \mathsf{Alive}$ (standard implication). McCain and Turner use causal rules of the form

$$\alpha \Rightarrow \beta \tag{3.1}$$

where α and β are formulas[3]. Informally, this rule expresses a relation of determinism between the states of affairs that make α and β true. To begin with, the causal laws are treated as inference rules; later in their article they are recast as rules called s-conditionals.

The standard derivability relation $\vdash$ of propositional logic is extended to take inference rules into account. Let Γ be a set of formulas and C be a set of inference rules. Γ is said to be *closed under* C if for every rule $\alpha \Rightarrow \beta \in C$, if $\alpha \in \Gamma$ then $\beta \in \Gamma$. For any formula α,

$$\Gamma \vdash_C \alpha$$

means that α belongs to the smallest set of formulas containing Γ that is closed with respect to propositional logic and closed under C.

[2] Causal laws here are what we call dependency laws.

[3] The $\Rightarrow$ symbol is similar to our $\gg$ sign.

How to get from one state to the next

The standard framework in which the problem of ramification is addressed is one in which background knowledge is given in the form of state constraints. For this framework, the problem was specified by Winslett [101] using the following definition.

Definition 3.3.1 ([71])
For any interpretation S, any explicit effect E, and any set B of formula constraints, $Res_B^W(E,S)$ is the set of interpretations S' such that

1. S' satisfies $E \cup B$, and
2. no other interpretation that satisfies $E \cup B$ differs from S on fewer atoms, where "fewer" is defined in terms of set inclusion. □

McCain and Turner propose the following reformulation of the above definition, in order to take inference rules into account.

Definition 3.3.2 ([71])
For any interpretation S, any explicit effect E, and any set C of inference rules, $Res_C^4(E,S)$ is the set of interpretations S' such that $S' = \{L : (S \cap S') \cup E \vdash_C L\}$. □

Declarative version

The trouble with the definition of Res_C^4 is that like Definition 3.3.1 it cannot be recast in semantic terms by replacing the derivability relation $\vdash$ with $\models$. To remedy this a conditional logic $\mathcal{C}_{\text{flat}}$ is defined. $\mathcal{C}_{\text{flat}}$ is an extension of S5 modal logic.

The vocabulary of the $\mathcal{C}_{\text{flat}}$ language consists of a fixed set of atoms. The formulas of the language are formed from its atoms and expressions of the form (3.1). Here expressions of the form (3.1) are called *s-conditionals*, and they may be read as: "the truth of α determines the truth of β".

A structure for a $\mathcal{C}_{\text{flat}}$ language is a pair (Ω, S), where Ω is a non-empty set of interpretations (of the atoms of the language), and S is an interpretation such that $\{S\} \in \Omega$.

For any set U of interpretations and any formula α of propositional logic, $U \models \alpha$ is an abbreviation for the expression $\forall S \in U.S \models \alpha$. Furthermore, for any set Γ of formulas of propositional logic, $U \models \Gamma$ is an abbreviation for the expression $\forall \alpha \in \Gamma.U \models \alpha$.

The definition of a structure (Ω, S) *satisfying* a formula α is as follows. For all formulas α and β (except in the last clause below, where α and β are formulas of propositional logic),

$$\begin{aligned}
(\Omega, S) &\models \alpha \text{ iff } \alpha \in S, \quad \text{if } \alpha \text{ is an atom,} \\
(\Omega, S) &\models \neg\alpha \text{ iff } (\Omega, S) \not\models \alpha, \\
(\Omega, S) &\models \alpha \wedge \beta \text{ iff } \ (\Omega, S) \models \alpha \text{ and } (\Omega, S) \models \beta, \\
(\Omega, S) &\models \alpha \Rightarrow \beta \text{ iff for all } U \in \Omega, \text{ if } U \models \alpha \text{ then } U \models \beta.
\end{aligned}$$

Let T be a set of formulas. A *model* of T is a structure that satisfies every formula in T. A model (Ω, S) is said to be *maximal* if there is no model (Ω', S') of C such that Ω is a proper subset of Ω'. We can now define a version of Res^4_C called $Res^{4'}_M$:

Definition 3.3.3 ([71])
Let C be a set of s-conditionals, and let $M = (\Omega, S'')$ be a maximal model of C. For any state S and explicit effect E, $Res^{4'}_M(E, S)$ is the set of states S' such that

$$S' = \{L : \forall U \in \Omega, \text{ if } U \models (S \cap S') \cup E \text{ then } U \models L\} \qquad \square$$

Example 3.3.2
To illustrate this formalism we encode the fact that a light (*Light*) is on only when two switches (*Up1* and *Up2*) are in the same position. We let the set of inference rules C be $\{(Up1 \leftrightarrow Up2) \Rightarrow Light, \neg(Up1 \leftrightarrow Up2) \Rightarrow \neg Light\}$, the explicit effect E be $\{Up1\}$, and choose the starting state S to be $\{\neg Up1,\ Up2, \neg Light\}$. The resulting set of states is $Res^{4'}_M(E, S) = \{\{Up1,\ Up2,\ Light\}\}$. $\square$

Observations about the formalism

As we can see, this formalism only deals with how to describe a relation between one state and the next. How it handles time and observations is not clearly specified, but it should be considered to be event-based. One weakness of the formalism is that it is completely declarative and provides no guidelines on how to compute $Res^{4'}_M$ in practice.

The language of the formalism is propositional logic, which limits the expressivity considerably. There is only one time-point in the left-hand side of the causal laws which means that there is no distinction between triggers and preconditions.

An interesting fact is that McCain and Turner present both a fixpoint and a non-fixpoint variant of the function that computes the successor state. The fixpoint version handles unstratified theories, but the non-fixpoint version does not. This is shown in Example 4.4.2.

3.3.3 Lin, 95

Another proposal using explicit causality was presented by Lin [67] at the same time as McCain and Turner [71]. Using a *Caused* predicate, directionality of causation can be encoded within the situation calculus framework. Lin later shows how this approach can be used to deal with nondeterminism [68].

Definitions

The usual situation calculus predicates and functions are used:

- The binary function do – for any action a and any situation s, $do(a, s)$ is the situation resulting from performing a in s.
- The binary predicate $Holds$ – for any propositional fluent p and any situation s, $Holds(p, s)$ is true if p holds in s.
- The binary predicate $Poss$ – for any action a and any situation s, $Poss(a, s)$ is true if a is possible (executable) in s.
- S_0 is the initial situation.

Time progresses through the execution of actions, that is, successive applications of the do function, leading to a branching time-structure.

In addition to the standard situation calculus notation a ternary predicate *Caused* is introduced. The first argument is a fluent, the second is a truth value, and the third is a situation. For example, $Caused(p, \mathsf{true}, s)$ denotes that the fluent p is caused to be true in situation s.

The following axioms are included to handle the *Caused* predicate:

$$Caused(p, \mathsf{true}, s) \rightarrow Holds(p, s), \tag{3.2}$$

$$Caused(p, \mathsf{false}, s) \rightarrow \neg Holds(p, s) \tag{3.3}$$

and the frame axiom

$$Poss(a, s) \rightarrow ((\exists v)\, Caused(p, v, do(a, s)) \rightarrow [Holds(p, do(a, s)) \leftrightarrow Holds(p, s)]) \tag{3.4}$$

From this the *pseudo successor state axiom* can be derived[4]:

$$\begin{aligned} Poss(a,s) \rightarrow \quad & \{Holds(p, do(a,s)) \leftrightarrow \\ & Caused(p, \mathsf{true}, do(a,s)) \vee \\ & Holds(p,s) \wedge \neg Caused(p, \mathsf{false}, do(a,s))\}. \end{aligned} \tag{3.5}$$

A formula $\Phi(s)$ is called a *simple state formula about* s if Φ does not mention *Poss*, *Caused* or any situation term other than s.

Procedure

The procedure of constructing a logical theory can be summarized as follows:

Step 1a: For each action $A(\vec{x})$, formalize the *direct effects* of A by axioms of the form:

$$\begin{aligned} Poss(A(\vec{x}), s) \quad & \rightarrow \\ & \Phi(s) \rightarrow Caused(F(\vec{y}), v, do(A(\vec{x}), s)), \end{aligned} \tag{3.6}$$

where F is a fluent name, and $\Phi(s)$ is a simple state formula about s

Step 1b: For each action $A(\vec{x})$, formalize the *explicit qualifications* of A by axioms of the form:

$$Poss(A(\vec{x}), s) \rightarrow \Phi(s), \tag{3.7}$$

where $\Phi(s)$ is a simple state formula about s.

Step 1c: Formalize all *causal rules* by axioms of the form:

$$\Phi(s) \wedge Caused(p_1, v_1, s) \wedge ... \wedge Caused(p_n, v_n, s) \rightarrow Caused(F(\vec{x}), v, s), \tag{3.8}$$

where F is a fluent, and $\Phi(s)$ is a simple state formula about s.

[4] This can be compared to the single, fixed, general frame axiom of Elkan [28]:

$$\begin{aligned} \forall a, s, p \quad & holds(p, do(s,a)) \leftrightarrow \\ & causes(a,s,p) \vee (holds(p,s) \wedge \neg cancels(a,s,p)). \end{aligned}$$

Step 1d: Formalize all other domain knowledge by axioms of the form $\forall s.C(s)$, where $C(s)$ is a simple state formula about s.

The above four steps yield a starting theory T.

Step 2: Circumscribe *Caused* in T with all other predicates fixed. Since this is done using Clark completion, the theory T has to be stratified (that is, that it must not contain causal cycles; see Section 4.4.4).

Step 3: Replace the occurrences of *Caused* in the pseudo successor state axiom (3.5) to get the form $Caused(F(\vec{x}), v, s) \leftrightarrow \Psi$, where Ψ is a formula not containing the predicate *Caused*.

Step 4: For each action A, maximize $Poss(A(\vec{x}), s)$ with $Holds(p, s)$ fixed but *Caused* and $Holds(p, s') \wedge s' \neq s$ allowed to vary.

Example 3.3.3
Below is a scenario with a causal law that states that shooting someone results in that he stops walking.

$$\begin{aligned}
Poss(start\text{-}walk, s) &\rightarrow Caused(walking, \mathsf{true}, do(start\text{-}walk, s)),\\
Poss(end\text{-}walk, s) &\rightarrow Caused(walking, \mathsf{false}, do(end\text{-}walk, s)),\\
Poss(shoot, s) &\rightarrow Caused(dead, \mathsf{true}, do(shoot, s)),\\
Poss(start\text{-}walk, s) &\rightarrow \neg walking(s),\\
Poss(end\text{-}walk, s) &\rightarrow walking(s),\\
dead(s) &\rightarrow Caused(walking, \mathsf{false}, s).
\end{aligned}$$

□

Observations about the formalism

Lin allows preconditions (Φ) and triggers in the form of conjunctions. The postcondition is limited to a literal. This means that his dependency laws cannot model nondeterminism in its current form. A statement of the type "if the coin is in the air then it will fall down on either a black or a white square" will be difficult to encode with this type of dependency law. Actions can have nondeterministic effects, however [68].

One interesting aspect of this method is that it is the first general approach to the ramification problem that does not use a fixpoint computation to determine the resulting states. It is also the first approach that discusses the stratification problem. The minimization policy used in Lin's method is

similar to the original PMON policy. This being the case, some minor extensions to the original PMON minimization policy and some additional macros in the surface language $\mathcal{L}$(ND) made it possible to model dependencies in TAL, we discuss this in Chapter 4.

3.3.4 Thielscher, 95

Thielscher regards the direct effects obtained after firing an action merely as a preliminary approximation to the resulting situation [97, 98, 99]. The resulting situation is computed by first applying direct effects and then performing additional post-processing steps that generate indirect effects relative to domain constraints specified as causal rules. For each action there is a set of states; each state represents a possible direct effect of the action. Following the direct effect of an action is a sequence of states representing the progression towards a state that is legal with respect to the domain constraints[5]. In order to "trigger" causal rules in the right order and direction, additional knowledge about the directionality of causation (*influence relationships*) has to be specified. These influence relationships together with the domain constraints are compiled into causal relations, which are then used to progress between states in a cascade.

Definition 3.3.4
Let $\mathcal{F}$ be a finite set of symbols called *fluent names*. A *fluent literal* is either a fluent name $f \in \mathcal{F}$ or its negation, denoted by $\overline{f}$. A set of fluent literals is *inconsistent* iff it contains some $f \in \mathcal{F}$ along with $\overline{f}$. A *state* is a maximal consistent set of fluent literals. □

The set of *fluent formulas* is inductively defined in the standard manner.

Definition 3.3.5
Let $\mathcal{F}$ be a set of fluent names. An *action law* is a triple $\langle C, a, E\rangle$ where C, called the *condition*, and E, called the *effect* are consistent sets of fluent literals; and a, called the *action name*, is a symbol not occurring in $\mathcal{F}$. If S is a state then an action description $\alpha = \langle C, a, E\rangle$ is *applicable* in S iff $C \subseteq S$. The *application* of α to S yields $(S \backslash C) \cup E$; this will be called the *direct effect* of firing the action a. □

Note that nondeterminism of actions is achieved through the fact that an action may be associated with several action laws.

[5] Sandewall's cascades (see Section 3.3.5) are similar to this.

Ramifications

Thielscher addresses the ramification problem by regarding the state resulting from the computation of the direct effects merely as an intermediate state, which requires additional computation accounting for possible *indirect* effects. More specifically, a single indirect effect is obtained according to a directed *causal* relation between two particular fluents.

Definition 3.3.6
Let $\mathcal{F}$ be a set of fluent names. A *causal relationship* is an expression of the form

$$\alpha \texttt{ causes } \beta \texttt{ if } \Gamma,$$

where Γ is a fluent formula and α and β are fluent literals. □

These causal relationships are computed from causal constraints and influence information as will be explained later. Formulas which have to be satisfied in all possible states in a domain are called the *domain constraints*.

Definition 3.3.7
Let (S, E) be a pair consisting of a state S and a set of fluent literals E. Then a causal relationship α `causes` β `if` Γ is *applicable* to (S, E) iff $\Gamma \wedge \overline{\beta}$ is true in S and $\alpha \in E$. Its application yields the pair (S', E') where $S' = (S \backslash \{\overline{\beta}\}) \cup \{\beta\}$ and $E' = (E \backslash \{\overline{\beta}\}) \cup \{\beta\}$. □

If $\mathcal{R}$ is a set of causal relationships, then $(S, E) \rightsquigarrow_{\mathcal{R}} (S', E')$ denotes the existence of an element in $\mathcal{R}$ whose application to (S, E) yields (S', E'). $(S, E) \overset{*}{\rightsquigarrow}_{\mathcal{R}} (S', E')$ expresses the fact that successive application of the causal relations in $\mathcal{R}$ to (S, E) yields (S', E').

Definition 3.3.8
Let $\mathcal{F}$ and $\mathcal{A}$ be sets of fluent and action names, respectively, $\mathcal{L}$ a set of action laws, $\mathcal{D}$ a set of domain constraints, and $\mathcal{R}$ a set of causal relationships. Furthermore, let S be a state satisfying $\mathcal{D}$ and a an action name. If there exists an action description $\langle C, a, E \rangle \in \mathcal{A}$ which is applicable with respect to S, then the state S' is a *successor state* iff

1. $((S \backslash C) \cup E, E) \overset{*}{\rightsquigarrow}_{\mathcal{R}} (S', E')$ for some E' and
2. S' satisfies $\mathcal{D}$. □

Some types of causality are difficult to express due to the fact that α and β in a causal relationship must be literals. Therefore Thielscher provides a way to compute causal relationships from domain constraints together with a binary relation $\mathcal{I} \subseteq \mathcal{F} \times \mathcal{F}$ called the *influence information*. If $(f_1, f_2) \in \mathcal{I}$ then this is intended to denote that a change of f_1's truth-value might possibly influence the truth-value of f_2.

The technical details of the transformation from influence information and domain constraints to causal relations can be found in Thielscher [99].

Example 3.3.4
Let us consider the well-known switch example. The light is on if and only if both switches are on. There is a domain constraint $sw_1 \wedge sw_2 \leftrightarrow light$. If directionality from left to right is required the causal relationship is $\mathcal{I} = \{(sw_1, light), (sw_2, light)\}$. The result becomes:

$$
\begin{array}{llll}
sw_2 & \texttt{causes} & light & \texttt{if}\ sw_1 \\
sw_1 & \texttt{causes} & light & \texttt{if}\ sw_2 \\
\overline{sw_2} & \texttt{causes} & \overline{light} & \texttt{if}\ \top \\
\overline{sw_1} & \texttt{causes} & \overline{light} & \texttt{if}\ \top
\end{array}
$$

□

Observations about the formalism

Thielscher requires that triggers and postconditions are literals. At a first glance this might seem to imply that Thielscher's language has the same expressibility as Lin's, but the fact that it is a fixpoint method together with the freedom of choice in what order the causal relationships are applied makes it possible to express nondeterminism in Thielscher's approach.

The "coin in the air" problem could be encoded like this (assuming *black* and *white* are false in the initial state):

$$
\begin{array}{llll}
air & \texttt{causes} & black & \texttt{if}\ \overline{white} \\
air & \texttt{causes} & white & \texttt{if}\ \overline{black}
\end{array}
$$

This would result in a state where either *black* or *white* is true, not both. This is due to the fact that when we have chosen to apply one of the causal relationships in the initial state, then, in the resulting state, it will be impossible to apply the others. The choice of which of the causal relationships should be applied is what leads to the nondeterminism.

Thielscher's approach is quite versatile. The restricted form of the causal relationship may seem limiting, but used correctly it is possible to handle

many important aspects of the ramification problem and related problems, which include unstratified theories, ramification as qualification, and some aspects of delays and concurrency.

3.3.5 Sandewall, 96

A method for assessment of ramification is proposed by Sandewall in [91]. The idea is that different approaches should be *assessed* with respect to an *underlying semantics* which formally defines the intended conclusions for a class of scenario descriptions. The purpose of the underlying semantics is to define the set of intended models, and thereby the set of intended conclusions, in a precise, simple and intuitively convincing fashion.

The method for handling ramification that serves as the basis on which other approaches are judged has much in common with Thielscher's method in the sense that it is basically a fixpoint approach. The details of the causal chains, however, are captured in the function N, thereby avoiding the need for a fixpoint computation.

Causal propagation semantics

Let $\mathcal{R}$ be the set of possible states in the world, formed as the cartesian product of the finite range sets of a finite number of *state variables.* Also, let $\mathcal{E}$ be the set of possible actions, and let the *main next-state function* $N(E, r)$ be a function from $\mathcal{E} \times \mathcal{R}$ to non-empty subsets of $\mathcal{R}$. The function N is intended to indicate the set of possible result states if the action E is performed when the world is in state r. The assumption $N(E, r) \neq \emptyset$ expresses the fact that every action E can always be executed in every starting state r.

A binary non-reflexive *causal translation relation* C between states is introduced; if $C(r, r')$ then r' is said to be a successor of r. A state r is said to be stable iff it does not have any successor. The set $\mathcal{R}_C$ of *admitted states* is chosen as a subset of $\mathcal{R}$ all of whose members are stable with respect to C.

Furthermore, an *action invocation relation* $G(E, r, r')$ is a relation where $E \in \mathcal{E}$ is an action, r is the state where the action E is invoked, and r' is a new state where the instrumental part of the action has been executed. For every E and r there must be at least one r' such that $G(E, r, r')$, that is, every action is always invocable.

An *action system* is a tuple $\langle \mathcal{R}, \mathcal{E}, C, \mathcal{R}_C, G \rangle$. For any state $r \in \mathcal{R}_C$, consider a state r_1 such that $G(E, r, r_1)$, and a sequence of states $r_1, r_2, ..., r_k$ where $C(r_i, r_{i+1})$ holds for every i between 1 and $k - 1$, and where $r_k \in \mathcal{R}_C$

is a stable state. Such a sequence is called a *transition chain*, and r_k is considered as a *result state* of the action E in the situation r.

The intention is that it shall be possible to characterize the resulting state r_k in terms of E and r, but without referring explicitly to the details of the intermediate states. The following assumptions (or something similar) are needed in order to make this work as intended.

Definition 3.3.9
If three states r, r_i and r_{i+1} are given, the pair r_i, r_{i+1} is said to *respect* r iff $(r_i(f) \neq r_{i+1}(f)) \rightarrow (r_i(f) = r(f))$, for any state variable f that is defined in $\mathcal{R}$. Then, an action system $\langle \mathcal{R}, \mathcal{E}, C, \mathcal{R}_C, G \rangle$ is said to be *respectful* iff for every $r \in \mathcal{R}_C$ and every $E \in \mathcal{E}$, r is respected by every pair r_i, r_{i+1} in every transition chain, and the last element of the chain is a member of $\mathcal{R}_C$. □

This condition amounts to a "write-once" or "single-assignment" property: if the action E is performed in state r, the world may go through a sequence of states, but in each step from one state to the next, there cannot be changes in state variables which have already been changed previously in the sequence, nor can there be any additional change in a state variable that has changed in the invocation transition from r to r_1. This definition can be compared to Lin's definition of stratified systems in [67], or the definitions of applicability and successor axiom by Thielscher (presented as Definition 3.3.7 and 3.3.8 in this thesis).

As a consequence of these definitions, if $\langle \mathcal{R}, \mathcal{E}, C, \mathcal{R}_C, G \rangle$ is a respectful action system, and $r \in \mathcal{R}_C, E \in \mathcal{E}$ and $G(E, r, r')$ holds for some r', then all transition chains that emerge from (E, r) are finite and cycle-free. The set of states will be denoted $N(E, r)$. The set $N(E, r)$ is derived from the elementary relations G and C in the action system.

Respectful action systems are intended to capture the basic intuitions of actions with indirect effects which are due to causation, as follows. Suppose the world is in a stable state r, and an action E is invoked. The immediate effect of this is to set the world in a new state, which is not necessarily stable. If it is not, then one allows the world to go through the necessary sequence of state transitions until it reaches a stable state. That *whole* sequence of state transitions is viewed as the action, and the resulting admitted state is viewed as the result state of the action.

Observations about the formalism

One has to remember that this is a method used for assessments. This makes a comparison with more detailed methods somewhat problematic. Some

general observations can, however, be made about the assessment method, for example that it handles causal cycles.

Sandewall uses a fixpoint-oriented approach, but the results of the cascades are captured by the N function. Using N, the approach can be considered to be a non-fixpoint method.

The fact that C is a relation between only two states and that change is not remembered makes it impossible to distinguish between preconditions and triggers. So, just as is the case in Geffner's and McCain and Turner's approaches, preconditions cannot be expressed in Sandewall's approach.

3.3.6 McCain and Turner, 97

The second formalism by McCain and Turner [72] is inspired by the work of Pearl ([80, 81]). The underlying propositional signature is described by three pairwise-disjoint sets: A set of *action names*, a nonempty set of *fluent names*, and a nonempty set of *time names* (corresponding to a segment of the integers). The atoms of the language are expressions of the forms a_t and f_t, where a, f, and t are action, fluent, and time names, respectively. Intuitively, a_t is true iff action a occurs at time t, and f_t is true only iff the fluent f holds at time t. A *fluent formula* is a propositional combination of fluent names.

A *causal law* is an expression of the form $\Psi \Rightarrow \Phi$, where Ψ and Φ are formulas of the underlying propositional language. Ψ is called the *antecedent* and Φ is called the *consequent* of the causal law. A causal law $\Psi \Rightarrow \Phi$ can mean both

- the fact that Ψ causes the fact that Φ, and
- necessarily, if Ψ then the fact that Φ is caused.

A *causal theory* is a set of causal laws.

An interpretation I for a propositional language is identified with the set of literals L such that $I \models L$. For every causal theory D and interpretation I, let

$$D^I = \{\Psi | \text{for some } \Phi, \Phi \Rightarrow \Psi \in D \text{ and } I \models \Phi\}.$$

An interpretation I is *causally explained* according to D if and only if $I = \{L : D^I \models L\}$, where L is understood to stand exclusively for literals.

A fluent has to have a cause for its value at each time-point. This is the case even if the fluent retain the value it had at the previous time-point, so in order to handle inertia we have to add a cause for it. The formula

$\sigma_t \wedge \sigma_{t+1} \Rightarrow \sigma_{t+1}$, where σ is a meta-variable ranging over I, expresses the fact that the fluents designated by the fluent formulas in I are subject to inertia. This states that if a fluent is inert then it has a cause to be inert.

Let D be a causal theory in which

1. the consequent of every causal law is a literal, and

2. every literal is the consequent of finitely many causal laws.

Literal completion of D means the classical propositional theory obtained by an elaboration of the Clark completion method [15], as follows: For each literal L in the language of D, include the formula $L \equiv (\Phi_1 \vee ... \vee \Phi_n)$, where $\Phi_1, ..., \Phi_n$ are the antecedents of the causal laws with consequent L.

Observations about the formalism

In order to apply the literal completion of a theory, McCain and Turner restrict the consequent of the casual laws to be a literal. Lifschitz [64] provides an extension of this formalism to non-propositional logic with arbitrary causal rules without any restrictions on the syntactic form of the pre- and postconditions. This transformation may introduce second-order quantifiers, but Lifschitz claims that these quantifiers can be eliminated in many cases.

McCain and Turner's approach only describes causal rules and it is not clear how observations and action invocations are handled. McCain and Turner provide a non-fixpoint method for computing the models of a theory, which indicates that they would have trouble handling unstratified theories.

An interesting feature of this approach is that the timeline represented by adding a subscript to the fluents allows delayed effects, although the authors mention this only in passing.

3.3.7 Denecker, Dupré and Belleghem, 97

A systematic solution to the ramification problem is presented by Denecker, Dupré and Belleghem [17, 10]. They begin by stating three requirements they want their solution to satisfy:

First, the dependent effect propagation rules (the causal rules) should not necessarily be coupled with the state constraints. To exemplify this Denecker et al. propose a counter that increases its value each time-point. It is problematic to construct such a counter if the causality is tightly connected to state constraints which only reference a single time-point.

Second, Denecker et al. define a precise general semantics for their effect rules. They show that many approaches to ramification can be seen as instances of the same generic idea: Interpreting the set of rules as a generalized form of *inductive definition*. However, the approaches differ in their ability to handle cyclicity. They state:

> In reasoning about actions an acyclicity condition can be justified because of the acyclic nature of causality, but we argue that this notion should not be imposed on the syntactic level. This is because an evident requirement in knowledge representation is the modularity of representation: it should be possible to describe the behavior of a system in terms of the behavior of its components, where each component *type* has a specific representation which is used for different instances, in the same system or in different systems. The model of a system should only be dependent on the types of all components and on their connections.

Third, they require their solution to be compact and natural.

Given these requirements Denecker et al. describe three increasingly general approaches. Only the last, most general approach will be briefly described here. The complete definitions of the approaches are found in Denecker et al. [17].

The general semantics approach

Let D be a domain of propositional symbols, including the symbols $\mathbf{t}$ and $\mathbf{f}$, denoting true and false. The elements of D are called atoms or positive literals, and the negation of elements of D are negative literals. The set $Defined(\mathcal{D})$ of atoms which occur in the head of a rule is called the set of defined atoms. Its complement $Open(\mathcal{D}) = \mathcal{D} \setminus Defined(\mathcal{D})$ is called the set of open atoms.

The basic language consists of the primitives *holds(l)* meaning that l is true in the initial state, *caus(l)* meaning that there exists a cause for changing l to true and *act(a)* denoting the invocation of an action a. The macro *init(l)* is reduced according to $init(l) \equiv caus(l) \wedge holds(\neg l)$.

Definition 3.3.10 ([17])
A $\mathcal{D}$-proof-tree $\mathcal{T}$ of $p \in D$ is a tree of literals of D with p as root such that:

- all leaves of $\mathcal{T}$ are open literals (possibly **t, f**) or defined negative literals;
- for each non-leaf node p with set of immediate descendants B, it holds that $p \leftarrow B \in \mathcal{D}$; hence p is a defined symbol in $\mathcal{D}$;
- $\mathcal{T}$ contains no infinite branches (hence it is free from cycles). □

The truth value of an atom is determined in the following way. If any proof tree has only true leaves, the atom is true, and if all proof trees have a false leaf, the atom is false. Since there is no guarantee that for a particular atom the truth values of all proof trees are known, a third truth value **u** is used to designate atoms of which the truth could not or has not yet been determined.

Definition 3.3.11 ([17])
Given a set of defined ground atoms $\mathbf{P}$, the set $\mathcal{V}_p$ of (3-valued) valuations on $\mathbf{P}$ is the set of all functions $\mathbf{P} \rightarrow \{\mathbf{t},\mathbf{f},\mathbf{u}\}$. On $\mathcal{V}_p$, a partial order $\leq_F$ is defined as the pointwise extension of the order $\mathbf{u} \leq_F \mathbf{t}$, $\mathbf{u} \leq_F \mathbf{f}$ □

Given a valuation $I \in \mathcal{V}_p$ and an interpretation O of the open atoms, for each $l \in \mathbf{P}$ we define its *supported value* with respect to I and O, denoted $SV_{I,O}(l)$, as the truth value proven by its best proof tree:

Definition 3.3.12 ([17])

- $SV_{I,O}(l) = \mathbf{t}$ if l has a proof tree with all leaves containing true facts with respect to I and O;
- $SV_{I,O}(l) = \mathbf{f}$ if each proof tree of l has a false fact with respect to I and O in a leaf;
- $SV_{I,O}(l) = \mathbf{u}$ otherwise. □

Definition 3.3.13 ([17])
The induction operator $\mathcal{PI}_{\mathcal{D},O} : \mathcal{V}_\mathbf{P} \rightarrow \mathcal{V}_\mathbf{P} : I \rightarrow I'$ is defined such that $\forall p \in \mathbf{P} : I'(p) = SV_{I,O}(p)$. □

It can be proven that this operator is monotonic and hence always has a least fixpoint $\mathcal{PI}_{\mathcal{D},O} \uparrow$ for a given interpretation O of the open atoms. This allows the general definition $\mathcal{D}$ with respect to O, denoted $\mathcal{I}_{\mathcal{D},O}$, as follows:

Definition 3.3.14 ([17])
Given $< \mathbf{P}, \mathcal{D}, O >$, $\mathcal{I}_{\mathcal{D},O} = \mathcal{PI}_{\mathcal{D},O} \uparrow$. □

Using this mechanism it is possible to compute the legal interpretations of a scenario.

Summary and comparison with TAL

The simple, short description of the general semantics used by Denecker et al. is that each atom should have a finite explanation, without any casual cycles. The definition is simple and powerful, and seems to handle a broad class of scenarios. Concurrency, delays, duration and complexity are briefly discussed, but it would be more interesting to see a thorough analysis of how these aspects are solved within the general semantics.

The paper discusses a number of problems resulting from too strong connections between causal laws and state constraints in certain approaches. These problems are not present in TAL, since TAL constraints can range over several states and the relaxation of inertia can be completely separated from state constraints. A construction like the counter example in Denecker et al. [17] can easily be modeled in TAL (for a slightly more complex counter, see for example the move_up procedure in the elevator example on page 140).

The paper also discusses the conflicts that can arise from defining rules such as

$$caus(p) \leftarrow \neg caus(p).$$

These types of nonsense statements cannot occur in TAL because causality does not exist as a first class citizen in our logic. We can only express causality through changing or observing change in the values of fluents. The closest we can get to the above expression is

$$\mathbf{dep}\ \neg([t]\neg p \wedge [t+1]p) \rightarrow I([t+1]p)$$

which is equivalent to $\forall t.[t]p$.

3.3.8 Shanahan, 99

A complete introduction to Shanahan's event calculus and the research area of reasoning about action and change can be found in Shanahan [93, 94].

The formalism's basic predicates are as follows. $Initiates(\alpha, \beta, \tau)$ means that the fluent β starts to hold after action α at time τ, $Terminates(\alpha, \beta, \tau)$ means that the fluent β ceases to hold after action α at time τ, $Releases(\alpha, \beta, \tau)$

means that the fluent β is not subject to inertia after action α at time τ, $Initially_P(\beta)$ means that the fluent β holds from time 0, $Initially_N(\beta)$ means that the fluent β does not hold from time 0, $Happens(\alpha, \tau)$ means that the action α occurs at time τ, and $HoldsAt(\beta, \tau)$ means that the fluent β holds at time τ. The axioms describing the relations between the predicates are the following:

$$HoldsAt(f,t) \leftarrow Initially_P(f) \wedge \neg Clipped(0,f,t)$$
$$HoldsAt(f,t_2) \leftarrow Happens(a,t_1) \wedge Initiates(a,f,t_1) \wedge t_1 < t_2 \wedge \neg Clipped(t_1,f,t_2)$$
$$Clipped(t_1,f,t_3) \rightarrow \exists a,t_2 \leftrightarrow \exists a,t_2[Happens(a,t_2) \wedge t_1 < t_2 \wedge t_2 < t_3 \wedge [Terminates(a,f,t_2) \vee Releases(a,f,t_2)]]$$
$$\neg HoldsAt(f,t) \leftarrow Initially_N(f) \wedge \neg Declipped(0,f,t)$$
$$\neg HoldsAt(f,t_2) \leftarrow Happens(a,t_1) \wedge Terminates(a,f,t_1) \wedge t_1 < t_2 \wedge \neg Declipped(t_1,f,t_2)$$
$$Declipped(t_1,f,t_3) \rightarrow \exists a,t_2[Happens(a,t_2) \wedge t_1 < t_2 \wedge t_2 < t_3 \wedge [Initiates(a,f,t_2) \vee Releases(a,f,t_2)]]$$

The conjunction of these axioms is denoted EC.

Added to this are state constraints and causal constraints. State constraints are *HoldsAt* formulas with a universally quantified time argument. The idea behind the causal law is, similar to the ideas in Pinto [84], to let an event be a carrier from cause to effect. For this the following four rules are needed:

$$Started(f,t) \leftrightarrow HoldsAt(f,t) \vee \exists a[Happens(a,t) \wedge Initiates(a,f,t)]$$
$$Stopped(f,t) \leftrightarrow \neg HoldsAt(f,t) \vee \exists a[Happens(a,t) \wedge Terminates(a,f,t)]$$
$$Initiated(f,t) \leftrightarrow Started(f,t) \wedge \neg\exists a[Happens(a,t) \wedge Terminates(a,f,t)]$$
$$Terminated(f,t) \leftrightarrow Stopped(f,t) \wedge \neg\exists a[Happens(a,t) \wedge Initiates(a,f,t)]$$

Example 3.3.5
The following is an example of a causal constraint:

$$Happens(LightOn,t) \rightarrow Stopped(Light,t) \wedge Initiates(Switch1,t) \wedge Initiated(Switch2,t) \qquad \square$$

Given a conjunction Σ of *Initiates, Terminates* and *Releases* formulas, a conjunction Δ of $Initially_P$, $Initially_N$, *Happens* and temporal ordering formulas,

a conjunction Ψ of state constraints, and the unique names axiom Ω for action fluents, the minimization policy becomes:

$$CIRC(\Sigma, \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}) \wedge$$
$$CIRC(\Delta, \mathit{Happens}) \wedge EC \wedge \Omega \wedge \Psi$$

Shanahan discusses the inability of his approach to handle vicious cycles (his name for causal cycles), but no solution to the problem is presented. It is possible that the method used to handle cycles presented in Section 4.6 can be applied to the event calculus.

The mechanism to handle the ramification problem in the event calculus is straightforward and adds only small changes to the formalism.

3.4 Conclusions

As we can see there are a wide variety of methods for handling side-effects, but it is also obvious that some general systematizations can be made. We can, for example, distinguish between methods that check and execute all dependency laws within one state (which we call non-fixpoint approaches) and methods that use a sequence of states to reach a fixpoint (called fixpoint approaches). The reason for this distinction is that non-fixpoint approaches have difficulties handling theories with causal cycles. Other important differences are how the approaches handle time, whether they use propositional logic or predicate logic, if dependency laws can be nondeterministic, and so on.

Figure 3.1 contains an overview of the approaches we have considered in this chapter.

	McCain Turner, 95 [71]	Lin [67]	McCain Turner, 97 [72]	Denecker et al. [10]
Logic	Propositional	Predicate	Propositional	Propositional
Precondition	Fluent change	Fluent value and change	Fluent value and change	Fluent value and change
Postconditions	Formula	Literal	Literal	Literal
Handles causal cycles	No	No	No	Yes
Time	Event-based	Event-based	Metric	Not specified
Durational actions	No	No	No	No

	Shanahan [94]	Thielscher [98]	Sandewall [91]	
Logic	Propositional	Predicate	Propositional	
Precondition	Fluent value and change	Fluent value and change	Fluent change	
Postconditions	Literal	Literal	Formula equivalent	
Handles causal cycles	No	Yes	Yes	
Time	Metric	Event-based	Event-based	
Durational actions	No	No	No	

Figure 3.1: Overview of the expressivity of other approaches.

Note that the "Precondition" row in the table represents if and how preconditions are triggered. Most approaches have causal laws that can be triggered both on fluents values and on fluent change but some are restricted to trigger only on change.

Figure 3.2 illustrates the approaches that have influenced the development of the TAL family the most.

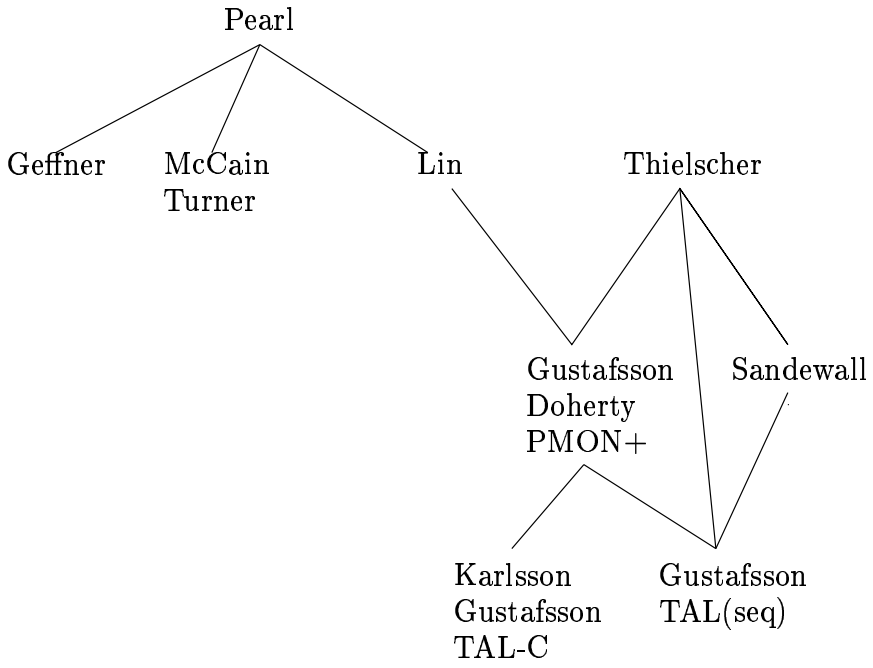

Figure 3.2: Brief overview of the influences between different methods of modeling causality.

Now that we have briefly surveyed a number of representative approaches to ramification, we proceed to present our own methods in the next chapter. Some of the following chapters contain additional comparisons to other approaches. These comparisons are more narrow in scope and only relate to the topics of those chapters.

Chapter 4

Ramification

This chapter introduces our approach to handling causal rules as presented in "Embracing occlusion in specifying the indirect effects of actions" [42]. Although the basic approach does not allow causal cycles, we also present two possible extensions for handling this.

4.1 Introduction

Although early versions of the PMON logic [20] were assessed correct for a broad class of action scenarios, they did not allow the modeling of indirect effects: All effects of an action, direct and indirect, had to be specified explicitly in a monolithic action description. In other words, those versions of PMON dealt with the simple frame problem in isolation and did not attempt to solve the ramification problem.

The first approach to solving the ramification problem within the PMON framework was presented by Doherty and Peppas [27], who extended the original PMON logic with a *frame* construct inspired by a similar construct used by Kartha and Lifschitz [50]. At any time-point, fluents in the frame were subject to inertia, while fluents not in the frame could vary freely, restricted only by domain constraints. Thus, the use of frame fluents, occluded frame fluents, and non-frame fluents in the resulting logic PMON(R) corresponded approximately to Kartha and Lifschitz's tripartite division of fluents into frame, frame released, and non-frame fluents.

In PMON(R), indirect effects affecting certain specific fluents could be modeled by placing those fluents outside the frame and using domain constraints to define their values in terms of the frame fluents. The extended logic was still characterized in terms of a circumscription axiom, but in con-

trast to the case for PMON, this axiom could not in the general case be reduced to a first-order formula.

Recently, causal minimization techniques have again become increasingly more popular. These techniques are based on introducing explicit causal predicates or causal rules to specify the indirect effects of actions [67, 68, 71, 98]. In this chapter, we will show how the base logic PMON can be extended with *causal rules* with little change to the existing formalism. In fact, causal rules in this context are really nothing more than macros in a surface language which when translated into the base language of PMON take advantage of the already existing predicate *Occlude*. As stated previously, the *Occlude* predicate has already been proven to be quite versatile in specifying actions with duration and indeterminate effects of actions. One of the benefits of using this approach to specify indirect effects of actions is that the original circumscription policy for the base logic PMON is left virtually intact. Consequently, the extended version, which we will call PMON(RCs), inherits the nice feature of allowing the reduction of any circumscribed action theory to a logically equivalent first-order theory provided the axiomatization of the chosen flow of time is first-order definable.

What is striking when investigating PMON(RCs) is the comparison with Lin's recent proposals for dealing with indirect effects of actions and indeterminate actions ([67, 68]). In fact, the *Cause* predicate and minimization policy used by Lin are virtually analogous with the use of the *Occlude* predicate and the minimization policy used in both the original PMON [90, 20], and the minor extension made in PMON(RCs).

There are a number of factors which makes a formal comparison between the two approaches difficult, although it should be mentioned that a study of the proper formal tools needed to do such comparisons was initiated in Doherty and Peppas [27]. In particular, we use a linear time structure, whereas Lin uses the situation calculus. In addition, Lin also deals with actions which may fail and with the qualification problem.[1] Finally, much of Lin's work deals with the automatic generation of successor state axioms as a means for computing entailment in the situation calculus framework.

In the rest of the chapter, we will do the following: Section 4.2 extends PMON by introducing causal and acausal constraints. Section 4.3 illustrates the approach with a number of examples. Section 4.4 discusses the problems associated with causal cycles, and Sections 4.5 and 4.6 present two very different approaches to solving those problems within the TAL framework.

[1] Although our article [42] did not cover those issues, they were later explored by Kvarnström and Doherty [52] with respect to the the TAL-C logic.

4.2 Extending PMON to handle Ramification

We will now proceed to the main topic of this chapter, that of extending PMON in order to deal with indirect effects of actions. The ramification problem states that it is unreasonable to explicitly specify all the effects of an action in the action specification itself. One would rather prefer to state the direct effects of actions in the action specification and use the deductive machinery to derive the indirect effects of actions using the direct effects of actions together with general knowledge of dependencies among fluents, specified as domain constraints. The dependencies specified using domain constraints do not necessarily have to be based solely on notions of physical causality.

The idea is that there is a certain tiered precedence for change where change in fluents of a certain class are dependent on changes of fluents in another class but not vice-versa. Which fluents have precedence over others is of course a domain dependent call and that information must be provided in some manner, for example, either explicitly in terms of causal rules, or perhaps implicitly in terms of partitioning fluents in classes such as framed, frame-released and non-framed as is the case with [50], and using a particular minimization policy. A particular domain policy might be based on physical causality, where the precedence is for causes to have a stronger inertia quota than their dependencies. On the other hand, when explaining effects, one might reverse the precedence. A good example is the domain constraint $light \equiv switch_1 \vee switch_2$ where causal flow is in the right–left direction, but one might equally well reverse the precedence for actions which turn a light on instead.

One of the difficulties in dealing with the ramification problem is the fact that a tension exists between solving the standard frame problem which requires minimizing change across the board, and attacking the ramification problem which requires relaxing minimization of change *just enough* to permit change for indirect effects, but only indirect effects that have a justification. As mentioned above, it is still an open issue as to what policies such justifications should be based on. Physical causality is simply one of several reasons one might set up dependencies between fluents. Sandewall [91] and Thielscher [98] have analyzed different policies for justifying dependencies between fluents.

The basis of our solution is a straightforward encoding of relaxing minimization of change just enough for the indirect effects of actions. This will be done by introducing causal rules in $\mathcal{L}(\mathrm{ND})$ which provide a means of ex-

pressing the directionality of dependencies between fluents and defining their translations into $\mathcal{L}$(FL) in terms of the *Occlude* predicate, which excludes the indirect effects from the non-change constraint in the nochange axiom Γ_{NCG}. We will begin by distinguishing between two types of domain constraints, *causal constraints* and *acausal constraints*, and then specifying both their representations in $\mathcal{L}$(ND) and translations into $\mathcal{L}$(FL), in terms of the *Occlude* predicate.

4.2.1 Causal and Acausal Constraints

In order to express causal constraints, we begin by defining causal relation expressions in $\mathcal{L}$(ND).

Definition 4.2.1
A *causal relation* is an expression in $\mathcal{L}$(ND) with the following form:

$$[t]\delta \gg [s]\gamma,$$

where both t and s are temporal terms in $\mathcal{L}$(FL), $t \leq s$, and δ and γ are fluent formulas. □

The intended meaning of a causal relation is "if the fluent formula δ is true at time-point t, then the fluent formula γ must be true at time-point s, and a change in γ due to this rule is legal with respect to the nochange premise".

Definition 4.2.2
A *causal constraint* is an expression in $\mathcal{L}$(ND) with the following form:

$$Q(\alpha \rightarrow [t]\delta \gg [s]\gamma) \text{ or } Q([t]\delta \gg [s]\gamma),$$

where α is a scenario formula in which for any temporal term t' in α, $t' \leq t$, and where Q is a sequence of quantifiers binding free variables of sorts $\mathcal{F}$ and $\mathcal{T}$. □

We call α the *precondition* for the causal constraint. The use of preconditions permits the representation of context dependent dependencies among fluents. Note also, that because the formalism uses explicit time and t and s may refer to different time-points, it is straightforward to represent delayed effects of actions using causal constraints. We will demonstrate this in the examples which follow in the next section. Causal constraints, also referred to as dependency constraints, will be labeled with the prefix "dep" in scenario descriptions in $\mathcal{L}$(ND).

The following expressions provide examples of conditionalized and non-conditionalized causal constraints:

dep_1 $\forall t([t]\mathsf{underwater} \rightarrow ([t]\mathsf{breathing} \gg [t + \mathfrak{t}_0]\neg\mathsf{alive}))$
dep_2 $\forall t([t]\neg\mathsf{alive} \gg [t]\neg\mathsf{walking})$

Acausal constraints describe relations between fluents without encoding any preference for dependency ordering. In a sense, a special class of acausal constraints is not really necessary. Technically, they could be defined as observations in a scenario which are observed at all time-points, but conceptually there is a difference.

Definition 4.2.3
An *acausal constraint* is an arbitrary quantified scenario formula in $\mathcal{L}$(ND). □

Acausal constraints will be labeled with the prefix "acc" in $\mathcal{L}$(ND). We call an acausal constraint where all fluents have the same temporal term a *static domain constraint*, while those with several temporal terms will be called *transition constraints*. The following expressions provide examples of static and transition constraints, respectively:

acc_1 $\forall t([t]\neg\mathsf{black} \vee [t]\neg\mathsf{white})$
acc_2 $\forall t([t]\neg\mathsf{alive} \rightarrow [t+1]\neg\mathsf{alive})$

4.2.2 Translation into $\mathcal{L}$(FL)

In the previous section, we defined a number of different domain constraint types in the surface language $\mathcal{L}$(ND). We will now provide a translation into $\mathcal{L}$(FL) and show that our informal intuitions regarding dependency preferences among constraints are formally encoded into $\mathcal{L}$(FL). Note that the only new symbol introduced when defining our domain expressions is the symbol $\gg$. The following macro-translation definition provides the proper translation for this relation.

Definition 4.2.4

$$[t]\delta \gg [s]\gamma \quad \stackrel{\text{def}}{=} \quad [([t]\delta \rightarrow [s]\gamma) \wedge \tag{4.1}$$
$$(([t-1]\neg\delta \wedge [t]\delta) \rightarrow [s]X(\gamma))], \tag{4.2}$$

where $t \leq s$ and where $[s]X(\gamma)$ denotes the occlusion of all restricted fluent terms in γ at s.[2] □

[2]For example, if γ is $\epsilon_1 C \epsilon_2$, where C is a logical connective, then $[s]X(\gamma)$ is $Occlude(s, \epsilon_1) \wedge Occlude(s, \epsilon_2)$.

This intermediate formula is then translated into $\mathcal{L}$(FL) in a manner similar to that described in the example in Section 2.3.

Example 4.2.1
The following causal constraint expression specified in $\mathcal{L}$(ND),

dep $\forall t[t](\neg \mathsf{alive} \gg \neg \mathsf{walking})$[3],

is translated into the following $\mathcal{L}$(FL) formula,

dep $\forall t[(\neg \mathit{Holds}(t, \mathsf{alive}) \rightarrow \neg \mathit{Holds}(t, \mathsf{walking})) \wedge$
$(\mathit{Holds}(t-1, \mathsf{alive}) \wedge \neg \mathit{Holds}(t, \mathsf{alive}) \rightarrow \mathit{Occlude}(t, \mathsf{walking}))]$.

□

The intuition behind the translation in Definition 4.2.4 is as follows: The first part of Definition 4.2.4, (4.1), represents the actual causal dependency. The formula $[t]\delta \rightarrow [s]\gamma$ forces γ to be true if δ is true. The second part (4.2) simply justifies the changes caused by the first part, but only in the appropriate causal direction. The formula $([t-1]\neg\delta \wedge [t]\delta) \rightarrow [s]X(\gamma)$, states that a change in γ caused by the rule is legal with respect to the nochange axiom.

4.2.3 PMON(RCs) Circumscription

The language of scenario descriptions $\mathcal{L}$(ND) has been extended for causal and acausal constraints and additional macro-translation definitions have been introduced which permit the translation of scenario descriptions with domain constraints into $\mathcal{L}$(FL). The final step will be to extend the circumscription policy used in PMON to accommodate these new changes. Let Γ_{ACC} and Γ_{DEP} denote the translations of the acausal and causal constraints into $\mathcal{L}$(FL), respectively. An action scenario Γ_C is now defined as $\Gamma_C = \Gamma_{FIL} \cup \Gamma_{CHG}$, where

$$\Gamma_{FIL} = \Gamma_{NCG} \cup \Gamma_{OBS} \cup \Gamma_{ACC} \cup \Gamma_{UNA} \cup \Gamma_T$$

$$\Gamma_{CHG} = \Gamma_{SCD} \cup \Gamma_{DEP}.$$

The new circumscription policy is defined as

$$\Gamma_{FIL} \wedge \mathit{Circ}_{SO}(\Gamma_{CHG}(\mathit{Occlude}); \mathit{Occlude}) \tag{4.3}$$

[3]In the rest of the chapter $[t](\delta \gg \gamma)$ will often be used instead of $[t]\delta \gg [t]\gamma$. This notation is only used in this chapter. Later we will use TAL-C instead and the I/R/X notation as presented in Chapter 5 and Appendix C.

Note that since Γ_{CHG} contains no negative occurrences of *Occlude*, and all other predicates are fixed, any action scenario in PMON(RCs) is provably reducible to a logically equivalent first-order theory. The first-order reduction applies to the extended scenarios without change.

4.3 Examples

In this section, we will consider two examples from the literature, the latter slightly modified from the original. These examples should help in acquiring both a conceptual and technical understanding of PMON(RCs).

Example 4.3.1
The walking turkey problem is a well-known ramification scenario and relatively straightforward to encode and solve using causal constraints. We will require one causal constraint stating that dead turkeys do not walk. One ramification of shooting a turkey is that it no longer walks and our theory should entail this indirect effect. The following action scenario description in $\mathcal{L}$(ND) describes the walking turkey problem:

Scenario Description 4.4
obs $[0]\mathsf{walking}$
occ $[2,4]\mathsf{Shoot}$
acs $[s,t]\mathsf{Shoot} \rightsquigarrow [s,t]\mathsf{alive} := F$
dep $\forall t.[t]\neg\mathsf{alive} \gg \neg\mathsf{walking}$

The corresponding translation into $\mathcal{L}$(FL) is,

obs $Holds(0, \mathsf{walking})$ (4.5)
scd $\neg Holds(4, \mathsf{alive}) \wedge \forall t(2 < t \leq 4 \rightarrow Occlude(t, \mathsf{alive}))$
dep $\forall t[(\neg Holds(t, \mathsf{alive}) \rightarrow \neg Holds(t, \mathsf{walking})) \wedge$
$(Holds(t-1, \mathsf{alive}) \wedge \neg Holds(t, \mathsf{alive}) \rightarrow Occlude(t, \mathsf{walking}))]$

The scenario is partitioned as:

$$\Gamma_{FIL} = \Gamma_{NCG} \cup \Gamma_{OBS} \cup \Gamma_{ACC} \cup \Gamma_{UNA} \cup \Gamma_{T},$$

where

$$\Gamma_{OBS} = \{Holds(0, \mathsf{walking})\}, \Gamma_{UNA} = \{\mathsf{alive} \neq \mathsf{walking}\}, \Gamma_{ACC} = \{\},$$

and

$$\Gamma_{CHG} = \Gamma_{SCD} \wedge \Gamma_{DEP},$$

where

$$\begin{aligned}
\Gamma_{SCD} &= \{\neg \mathit{Holds}(4, \mathsf{alive}) \wedge \forall t(2 < t \leq 4 \rightarrow \mathit{Occlude}(t, \mathsf{alive}))\}, \\
\Gamma_{DEP} &= \{\forall t[(\neg \mathit{Holds}(t, \mathsf{alive}) \rightarrow \neg \mathit{Holds}(t, \mathsf{walking})) \wedge \\
&\quad (\mathit{Holds}(t-1, \mathsf{alive}) \wedge \neg \mathit{Holds}(t, \mathsf{alive}) \rightarrow \mathit{Occlude}(t, \mathsf{walking}))]\}.
\end{aligned}$$

Circumscribing *Occlude* in Γ_{CHG} results in the following definition, which can be generated using either our algorithm [26] or standard predicate completion:

$$\begin{aligned}
\forall t, f.(\mathit{Occlude}(t, f) \quad &\leftrightarrow \quad ((f = \mathsf{alive} \wedge 2 < t \leq 4) \vee \\
&\quad (f = \mathsf{walking} \wedge \mathit{Holds}(t-1, \mathsf{alive}) \wedge \neg \mathit{Holds}(t, \mathsf{alive}))))
\end{aligned}$$

For readability, we list the complete translation of the scenario in $\mathcal{L}(\mathrm{FL})$, with the definition of *Occlude* derived via circumscription:

$$\begin{aligned}
&\forall f, t(\mathit{Holds}(t, f) \oplus \mathit{Holds}(t+1, f) \rightarrow \mathit{Occlude}(t+1, f)) \wedge \\
&\mathit{Holds}(0, \mathsf{walking}) \wedge \\
&\mathsf{alive} \neq \mathsf{walking} \wedge \\
&\neg \mathit{Holds}(4, \mathsf{alive}) \wedge \\
&\forall t[(\neg \mathit{Holds}(t, \mathsf{alive}) \rightarrow \neg \mathit{Holds}(t, \mathsf{walking})) \wedge \\
&\forall t, f(\mathit{Occlude}(t, f) \leftrightarrow ((f = \mathsf{alive} \wedge 2 < t \leq 4) \vee \\
&\qquad (f = \mathsf{walking} \wedge \mathit{Holds}(t-1, \mathsf{alive}) \wedge \neg \mathit{Holds}(t, \mathsf{alive}))))
\end{aligned}$$

There are two classes of preferred models for this scenario due to the fact that the shoot action has an extended duration and its effects may occur at time point 3 or 4:

$$\begin{aligned}
&[0, 2]\mathsf{alive} \wedge \mathsf{walking} \\
&[3, \infty]\neg\mathsf{alive} \wedge \neg\mathsf{walking}
\end{aligned}$$

and

$$\begin{aligned}
&[0, 3]\mathsf{alive} \wedge \mathsf{walking} \\
&[4, \infty]\neg\mathsf{alive} \wedge \neg\mathsf{walking}
\end{aligned}$$

□

Example 4.3.2
The extended stuffy room problem is based on the original problem due to Winslett [101], where a room gets stuffy if both ducts providing fresh air get blocked. We extend it by introducing a box which requires the use of chained causal rules, and by encoding a delayed effect of an action by using one of the causal rules.

In this scenario, there are two causal constraints. The first asserts that duct_2 is blocked if something is put on top of it (loc_2). The second asserts that the room gets stuffy two time-points after both ducts are blocked. Between time-points 2 and 5 a box is moved to location loc_2. The action scenario is represented in $\mathcal{L}$(ND) as:

Scenario Description 4.6

obs $[0](\mathsf{blocked}(\mathsf{duct}_1) \wedge \neg\mathsf{blocked}(\mathsf{duct}_2) \wedge \mathsf{at}(\mathsf{box}, \mathsf{loc}_1) \wedge \neg\mathsf{stuffy})$

occ $[2,5]\mathsf{move}(\mathsf{box}, \mathsf{loc}_1, \mathsf{loc}_2)$

acs $\forall x, l1, l2([s,t]\mathsf{move}(x, l1, l2) \rightsquigarrow [s,t]\mathsf{at}(x, l1) := F \wedge [s,t]\mathsf{at}(x, l2) := T)$

dep_1 $\forall t, x[t](\mathsf{at}(x, \mathsf{loc}_2) \gg \mathsf{blocked}(\mathsf{duct}_2))$

dep_2 $\forall t([t](\mathsf{blocked}(\mathsf{duct}_1) \wedge \mathsf{blocked}(\mathsf{duct}_2)) \gg [t+2]\mathsf{stuffy})$

Translating into $\mathcal{L}$(FL) and circumscribing results in a theory equivalent to the following:

$$
\begin{aligned}
&\mathit{Holds}(0, \mathsf{blocked}(\mathsf{duct}_1)) \wedge \neg\mathit{Holds}(0, \mathsf{blocked}(\mathsf{duct}_2)) \wedge \\
&\mathit{Holds}(0, \mathsf{at}(\mathsf{box}, \mathsf{loc}_1)) \wedge \\
&\neg\mathit{Holds}(0, \mathsf{stuffy}) \wedge \\
&\neg\mathit{Holds}(5, \mathsf{at}(\mathsf{box}, \mathsf{loc}_1)) \wedge \\
&\mathit{Holds}(5, \mathsf{at}(\mathsf{box}, \mathsf{loc}_2)) \wedge \\
&\forall t, x(\mathit{Holds}(t, \mathsf{at}(x, \mathsf{loc}_2)) \rightarrow \mathit{Holds}(t, \mathsf{blocked}(\mathsf{duct}_2))) \wedge \\
&\forall t(\mathit{Holds}(t, \mathsf{blocked}(\mathsf{duct}_1)) \wedge \mathit{Holds}(t, \mathsf{blocked}(\mathsf{duct}_2)) \rightarrow \\
&\qquad \mathit{Holds}(t+2, \mathsf{stuffy})) \wedge \\
&\forall f, t(\mathit{Holds}(t, f) \oplus \mathit{Holds}(t+1, f) \rightarrow \mathit{Occlude}(t+1, f)) \wedge
\end{aligned}
$$

$$
\begin{array}{l}
\forall t, f(\\
\quad \mathit{Occlude}(t, f) \leftrightarrow \\
\qquad (2 < t \leq 5 \wedge f = \mathsf{at}(\mathsf{box}, \mathsf{loc}_1)) \vee \\
\qquad (2 < t \leq 5 \wedge f = \mathsf{at}(\mathsf{box}, \mathsf{loc}_2)) \vee \\
\qquad \forall x(\neg \mathit{Holds}(t-1, \mathsf{at}(x, \mathsf{loc}_2)) \wedge \mathit{Holds}(t, \mathsf{at}(x, \mathsf{loc}_2)) \wedge \\
\qquad\quad f = \mathsf{blocked}(\mathsf{duct}_2)) \vee \\
\qquad (\neg[\mathit{Holds}(t-3, \mathsf{blocked}(\mathsf{duct}_1)) \wedge \mathit{Holds}(t-3, \mathsf{blocked}(\mathsf{duct}_2))] \wedge \\
\qquad\quad \mathit{Holds}(t-2, \mathsf{blocked}(\mathsf{duct}_1)) \wedge \mathit{Holds}(t-2, \mathsf{blocked}(\mathsf{duct}_2)) \wedge \\
\qquad\quad f = \mathsf{stuffy}))
\end{array}
$$

In addition, the unique name axioms are included, but not listed here. The theory correctly entails that if something is placed on top of duct_2 while duct_1 is blocked, then the room will become stuffy two time-points later. In the current axiomatization, the room would remain stuffy forever even after one of the ducts became free. This is because the nochange axiom prevents the room from becoming unstuffy without reason. In the current theory, there is no axiom which states otherwise. A rule stating that the room is stuffy only when both ducts are blocked can be added without difficulty:

dep_3 $\forall t([t]\neg(\mathsf{blocked}(\mathsf{duct}_1) \wedge \mathsf{blocked}(\mathsf{duct}_2)) \gg [t+2]\neg\mathsf{stuffy})$.

This example demonstrates both the use of casual chaining and how causal constraints can be used to specify delayed effects of actions. □

4.4 Causal Cycles

This section considers the problem posed by non-stratified theories (theories with causal cycles) and why non-fixpoint approaches have difficulties in handling them. After that we present two solutions to the stratification problem. The first solution, in Section 4.5, contains a modification of TAL, transforming it into a fixpoint approach similar to Thielscher's and Sandewall's solutions. The second solution, in Section 4.6, is based on an unpublished article by Gustafsson and Kvarnström. That approach does not need any modification to TAL since it works by moving the cycle detection into the surface language. This allows us to experiment with different types of cycle handling without having to change the underlying logic. It also shows that correct cycle handling can be done without the need to introduce fixpoint mechanisms.

Before the solutions are presented, let us begin by defining causal cycles and examining the nature of the problems associated with theories containing causal cycles.

4.4.1 Causal Cycles

The following definition of causal cycles is inspired by the definition of stratification by Lin [67].

Definition 4.4.1
Let T be a set of dependency laws. A causal cycle in T is a sequence of fluents $\langle f_0, f_1, \ldots, f_n \rangle$ such that $f_0 \Rightarrow f_1 \wedge f_1 \Rightarrow f_2 \wedge \ldots \wedge f_{n-1} \Rightarrow f_n \wedge f_n \Rightarrow f_0$, where for any fluents f and f', $f \Rightarrow f'$ if there is a dependency relation in T such that f appears in the trigger and f' appears, not delayed, in the postcondition in the dependency relation. □

For example, a theory containing the dependency laws $[t](\mathsf{dead} \gg \neg\mathsf{alive})$ and $[t](\neg\mathsf{alive} \gg \mathsf{dead})$ contains the causal cycle $\langle \mathsf{dead}, \mathsf{alive} \rangle$.

4.4.2 Causal Cycles in Non-Fixpoint Approaches

The existence of causal cycles in a theory causes problems for the non-fixpoint approaches to ramification discussed in this thesis, including Lin's approach and the PMON(RCs) approach discussed in the previous sections. The principle behind these approaches is as follows:

1. Let S be the current state.
2. Let W be the set of all possible states.
3. Remove those states from W that cannot be legal successors of S, considering fired actions, dependency laws, the inertia assumption and possibly other criteria.
4. The resulting set W contains all legal successor states with respect to S.

If a theory contains causal cycles, this procedure is not necessarily strong enough to filter out all unwanted models. For this reason, we have previously restricted our logic to deal only with stratified theories, where causal cycles do not occur. The reason for the problems associated with causal cycles will be explained using a small example.

Example 4.4.1
Consider a scenario description that contains two fluents alive and dead. If the fluents should be "synchronized" in the sense that if alive becomes false then it causes dead to become true and vice versa, one could imagine that this could be achieved using two dependency laws as in the example below.

Scenario Description 4.8
obs_1 $[0]\mathsf{alive} \wedge \neg\mathsf{dead}$
occ_1 $[5,6]\mathsf{Shoot}$
acs_1 $[t_1,t_2]\mathsf{Shoot} \rightsquigarrow [t_1,t_2]\mathsf{alive} := \mathsf{F}$
dep_1 $\forall t\ ([t]\neg\mathsf{alive} \gg [t]\mathsf{dead})$
dep_2 $\forall t\ ([t]\mathsf{dead} \gg [t]\neg\mathsf{alive})$

The intuitive conclusions from this scenario description would be that alive and dead remain unchanged until the Shoot action is executed. At 6, alive should become false and (by dep_1) dead should become true, and the fluents should retain the same values in $[6,\infty)$.

However, consider an interpretation where alive becomes false and dead becomes true at 1. Since there is a suitable change in alive, dependency law dep_1 is triggered, which occludes dead at 1, allowing dead to change values. Similarly, since there is a suitable change in dead, dependency law dep_2 is triggered, which occludes alive at 1, allowing alive to change values. Since both features are occluded, the inertia assumption (encoded in the nochange axiom) is not violated, and all observations and dependency constraints are satisfied. Thus, this interpretation is a valid model.

This means that it is impossible to infer that alive is true at time-point 1. □

This problem only arises for non-fixpoint approaches. The reason for this is that all ramifications take effect in the same state. In non-fixpoint approaches, the transitivity of dependency laws is immediate, whereas fixpoint approaches (discussed in the following subsection) have transitivity over a sequence of states. This means that in non-fixpoint approaches the postcondition of a dependency law may very well directly influence its own trigger. In fixpoint approaches the postcondition always takes effect in another (later) state than the trigger; this procedure is repeated until a fixpoint has been reached.

Of course, it is possible to argue that causal cycles are an artificial problem, and that if a certain scenario can be modeled with causal cycles it can be re-modeled without causal cycles, since causality is acyclic in nature. But this usually requires more detailed, complicated models of the world, and since

we are interested in common-sense reasoning, we often want to abstract away from such details and model causal relations that are not necessarily directly grounded in physical laws of nature, relations that may sometimes be cyclic. In addition it places an additional burden on the knowledge engineer.

4.4.3 Causal Cycles in Fixpoint Approaches

The general idea behind the fixpoint approaches to compute the successor states of state S is as follows:

1. Fire the action; call the resulting set of states S.
2. Let $R = \emptyset$.
3. For each $s \in S$ do:
   (a) If none of the dependency laws can be applied, then add s to R.
   (b) Otherwise, let W be the set of resulting states where at least one of the dependency laws has been applied to s.[4] [5] If $W \neq \emptyset$, then recursively apply step 3 to each w in W.
4. R is the set of successor states.

Thielscher [98] uses a fixpoint-oriented method. McCain and Turner [71] define both a fixpoint and a non-fixpoint approach. Sandewall [91] uses a fixpoint-oriented approach, but the results of the cascades are captured by the N function. Using N the approach could be considered as a non-fixpoint method.

Example 4.4.2
As an example that illustrates the difference between the two types of approaches, consider McCain and Turner's non-fixpoint method (Definition 3.3.3) and their fixpoint method (Definition 3.3.2) for a non-stratified theory. We will see that the fixpoint method provides the intended solution, whereas the non-fixpoint method results in models with counter-intuitive spontaneous change.

Let us consider a world with three fluents p, q and r, an initial state $S = \{\neg p, \neg q, \neg r\}$, an explicit effect $E = r$, and two causal laws $p \Rightarrow q$ and

[4] W is a set of states, since both the choice of dependency laws and the dependency laws themselves may be nondeterministic.

[5] We assume that all fluents not affected by the applied dependency laws remain inert in the states in W.

$q \Rightarrow p$. The intuitive result is $\{\neg p, \neg q, r\}$, since the value of r is changed and p and q should be unaffected by the action. The problem is that non-fixpoint methods in addition to the above result give the additional result $\{p, q, r\}$.

We begin by examining the fixpoint approach: Suppose $S_1' = \{\neg p, \neg q, r\}$. This means that $(S \cap S_1') \cup E = \{\neg p, \neg q, r\}$ and since $\{L : \{\neg p, \neg q, r\} \vdash_C L\} = S_1'$, S_1' must be a solution.

Now suppose $S_2' = \{p, q, r\}$, $(S \cap S_2') \cup E = \{r\}$. We get $\{L : \{r\} \vdash_C L\} \neq S_2'$, so S_2' is not a solution. So we see that $Res_C^4(E, S) = \{S_1'\}$ which is the intuitive result.

The interesting point is to examine $Res_M^{4'}(E, S)$ and see if it gives S_1' but not S_2'. The interesting interpretations are $U_1 = \{\neg p, \neg q, r\}$ and $U_2 = \{p, q, r\}$

When we try S_1' we see that $U_1 \models \{\neg p, \neg q, r\}$ and $\{L : U_1 \models L\} = S_1'$ so S_1' is a solution. But the surprise comes when we try S_2': $U_2 \models \{r\}$ and $\{L : U_2 \models L\} = S_2'$ so according to this $Res_M^{4'}(E, S) = \{S_1', S_2'\}$ which is not what we expected. □

4.4.4 Stratified Theories

In essence, a theory is stratified if it contain no causal cycles. The following is a slightly modified definition taken from Lin [67]:

Definition 4.4.2
We say that a set T of dependency laws is *stratified* if it contains no causal cycles (Definition 4.4.1). □

This means that a theory with the dependency laws $[t](\mathsf{dead} \gg \neg\mathsf{alive})$ and $[t](\neg\mathsf{alive} \gg \mathsf{dead})$ is not stratified.

This is the definition of stratification used in this thesis. An alternative, stricter definition of stratification is to require the fluents to be ordered in layers such that no fluent depends on any fluent in a higher layer.

We will now move on to describing the first solution to handling non-stratified theories, TAL(seq). The second solution is presented in Section 4.6.

4.5 Breaking Causal Cycles: I

We have seen that TAL with side-effects handles a broad class of problems. However, we have also seen that if the theories are not stratified, then TAL does not provide intuitive results.

Thielscher [98] solves this problem by viewing the effects caused by actions merely as an approximation of the final result. The complete result is reached by going through a sequence of states. Sandewall [91] also uses sequences of approximate states to reach a stable state.

The intention in this section is to combine the idea of sequences of states from Thielscher and Sandewall with the approach in TAL [42, 21]. This new combination will handle both stratified and non-stratified theories and will have the following advantages:

- We retain all the previous expressivity of TAL, including delayed effects, nondeterminism of both actions and dependency laws and the ability to use actions with extended duration.
- We build on existing work, using first-order predicate logic.
- The method is a declarative description of a fixpoint approach.

TAL(seq) should be viewed as a proposal for how TAL could be modified to handle both stratified and non-stratified theories, not as a complete formalism since many of the details which would make it so have been left out.

4.5.1 Three Types of States

We distinguish between three types of states, as illustrated in Figure 4.1. A *connecting state* at time t is the state resulting from all effects having taken place strictly before time t. The *primary-result state* at time-point t is the only state that is directly influenced by actions at t. All other states are called *dependent states*; these states are only influenced by dependency laws. For each time-point, a *cascade* is a chain of states beginning with a connecting state followed by a primary-result state followed by a chain of dependent states.

Since there is an explicit timeline in TAL, each time-point has to have its own cascade. This is an important difference between TAL(seq) and the formalisms developed by Thielscher and Sandewall.

To be able to distinguish between the states in the cascade, the *Holds* and *Occlude* predicates need an extra argument besides the fluent, the value and the time-point. Therefore we introduce an additional argument of sort $\mathcal{S}$ (the non-negative integers). It represents the number in the cascade (called the *cascade number*). This argument will be placed between the time-point and the fluent name.

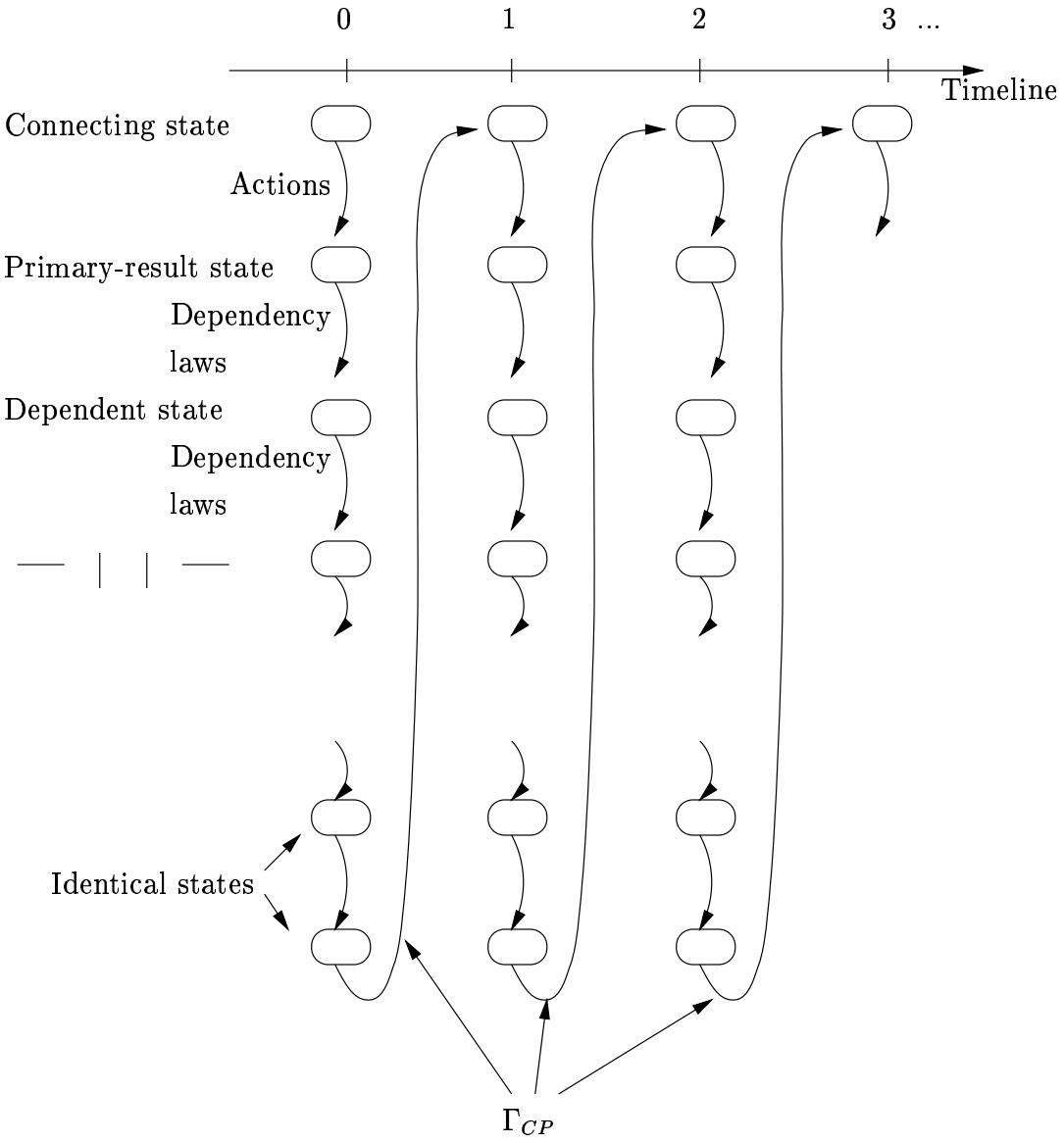

Figure 4.1: The relation between the different types of states.

The following shorthand notation will sometimes be used:

$$\begin{aligned} \{t,i\}f = v &\stackrel{\text{def}}{=} Holds(t,i,f,v) \\ \{t,i\}\neg f = v &\stackrel{\text{def}}{=} \neg Holds(t,i,f,v) \end{aligned}$$

If f is a fluent then

$$\begin{aligned} \{t,i\}\mathcal{X}(f) &\stackrel{\text{def}}{=} Occlude(t,i,f) \\ \{t,i\}\neg\mathcal{X}(f) &\stackrel{\text{def}}{=} \neg Occlude(t,i,f). \end{aligned}$$

If α is a formula then $\{t,i\}\mathcal{X}(\alpha)$ and $\{t,i\}\neg\mathcal{X}(\alpha)$ denote the conjunction of occlusion of all fluents in α.

The connecting state

Each cascade begins with a connecting state. This state is determined by the cascade of the preceding time-point as will be explained in Definition 4.5.3. The cascade number for this state is 0, so the fact that fluent f holds in the connecting state at time-point 3 will be denoted $Holds(3,0,f)$.

References to values of fluents at a given time-point t should be to the connecting state on the following time-point $t+1$ since we want to know what the value is after all dependency laws have taken effect.

The primary-result state

The primary-result state follows immediately after the connecting state. This is the state that has resulted when all direct effects of actions and delayed effects have been taken into consideration.

The translation of the reassignment macro (See Section A.4) has to be changed for this purpose.

Definition 4.5.1

$$\begin{aligned} Tran([t,t']f_k(\omega_1,...,\omega_{i_k}) := v) &\stackrel{\text{def}}{=} \\ &Holds(t',1,f_k(\omega_1,...,\omega_{i_k}),v) \wedge \\ &\forall t''\ (t < t'' \leq t' \rightarrow Occlude(t'',1,f_k(\omega_1,...,\omega_{i_k}))) \end{aligned} \tag{4.9}$$

□

Definition 4.5.5 describes how the dependency laws affect the primary-result state.

The dependent states

The primary-result state is followed by a chain of dependent states. This chain is generated by the instantaneous dependency laws as will be shown in Definition 4.5.5. The idea behind the approach is that when we can no longer apply any of the dependency laws, a stable state has been reached. The values of all fluents in this stable state are copied to the connecting state at the following time-point, as shown in Definition 4.5.3. This is the reason it is called a connecting state: It connects the fixpoint state in a cascade with the first state in the following cascade.

4.5.2 Defining the Behavior of Cascades

This section discusses the axioms needed for inertia in ramification sequences and for connecting the last state in a cascade to the following connecting state. Thereafter the syntactical definition of a dependency law is given together with the translation into our logic.

Definition 4.5.2
The nochange axiom in TAL(seq), Γ_{NCG2}, is [6]

$$\forall t \geq 0, s \geq 0, f[(\{t,s\}f \oplus \{t,s+1\}f) \rightarrow \\ (\{t,s+1\}\mathcal{X}(f) \wedge \forall s' \geq 0[s' \leq s \rightarrow \{t,s'\}\neg\mathcal{X}(f)])],$$

where the $\oplus$ operator denotes exclusive-or. □

The idea is straightforward: Only allow a fluent to change value the first time it is influenced in a cascade. There are two reasons for this restriction. The first is that it prohibits infinite cascades, provided that the number of fluents is finite. If n is the number of fluents, then a cascade must have reached a fixpoint before, or when it reaches, the dependent state number n. The second reason is that a cascade is considered to be instantaneous, so no time progresses between the primary-result state and the following connecting state. It would be counter-intuitive to allow a fluent to change value several times without at least some time elapsing. This closely resembles the respectfulness criterion used by Sandewall [91]. In this sense, the numerical indicators provided in cascades should be interpreted as indexes rather than time-points in a temporal progression.

The second axiom needed is for copying a stable state to the following connecting state.

[6] Like the old nochange axiom Γ_{NCG}, the new axiom Γ_{NCG2} is not part of the theory that is minimized. Thus, the negative occurrences of the *Occlude* predicate does not pose a problem for reduction.

Definition 4.5.3
The axiom for copying stable fluents, Γ_{CP}, is:

$$\forall t \geq 0, s \geq 0\ [\forall f(\{t,s\}f \leftrightarrow \{t,s+1\}f) \rightarrow \forall f(\{t,s\}f \leftrightarrow \{t+1,0\}f)] \qquad \square$$

If all fluents remain unchanged between the states at cascade number s and $s+1$, a stable state has been reached, and all fluents are copied to the following connecting state.

Dependency laws

Dependency laws can be divided into *causal dependency laws* and *explanatory dependency laws*. Causal dependency laws are of the kind "A causes B", for example "turning the switch causes the lamp to go on". Explanatory dependency laws work in the opposite direction, "If B is true, the explanation is that A is true", for example "if the lamp is on, the explanation must be that the switch has been turned". Causal and explanatory dependency laws will be handled in the same way since the difference is conceptual rather than technical. A more thorough discussion of this can be found in [72].

Definition 4.5.4
A dependency law in the surface language is an expression of the form

$$\forall t \geq 0([t]\alpha \rightarrow ([\tau_1(t)]\delta \gg [\tau_2(t)]\gamma)),$$

where α, δ and γ are fluent formulas, t, $\tau_1(t)$ and $\tau_2(t)$ are of type $\mathcal{T}$ and $t \leq \tau_1(t) \leq \tau_2(t)$ for all t. $\square$

The intended meaning of the above definition is: "If α is true at time-point t, then δ at time-point $\tau_1(t)$ influences formula γ to become true at time-point $\tau_2(t)$".

If $\tau_1(t) < \tau_2(t)$, the dependency law is called a *delayed dependency law*, and otherwise it is called an *instantaneous dependency law*. The case when $\tau_1(t) > \tau_2(t)$ is considered illegal.

Definition 4.5.5
A dependency law $[t]\alpha \rightarrow ([\tau_1(t)]\delta \gg [\tau_2(t)]\gamma)$ in $\mathcal{L}$(ND) is translated to the following formula in $\mathcal{L}$(FL):

$$
\begin{aligned}
&((\{t+1,0\}\alpha \wedge \tau_1(t) < \tau_2(t)) \rightarrow \\
&\quad [(\{\tau_1(t)+1,0\}\delta \rightarrow \{\tau_2(t),1\}\gamma) \wedge \\
&\quad (\{\tau_1(t),0\}\neg\delta \wedge \{\tau_1(t)+1,0\}\delta \rightarrow \\
&\quad\ \{\tau_2(t),1\}\mathcal{X}(\gamma))]) \wedge \qquad (4.10)\\
&\tau_1(t) = \tau_2(t) \rightarrow [\\
&\quad (\{t,0\}\alpha \wedge \{\tau_1(t),0\}\delta \rightarrow \{\tau_1(t),0\}\gamma) \qquad (4.11)\\
&\quad\ \wedge \\
&\quad \forall s \geq 0[\\
&\quad\ (\{\tau_1(t),s\}\neg\delta \wedge \{\tau_1(t),s+1\}\delta \wedge \\
&\quad\ \{\tau_1(t),s+1\}\alpha) \rightarrow \\
&\quad\ (\{\tau_1(t),s+2\}\gamma \wedge \{\tau_1(t),s+2\}\mathcal{X}(\gamma))]] \qquad (4.12)
\end{aligned}
$$

The first part (4.10) takes care of delayed dependency laws, and ensures that they are effectuated in the primary-result state, first with respect to the assigned value and then with respect to occlusion. For instantaneous dependency laws, (4.11) ensures that all connecting states are legal with respect to the dependency law. So, even if the intermediate states in a cascade violate some of the domain constraints, we are sure to end in a legal state. Sentence (4.12) enforces the actual change in the cascade.

Translation and minimization

The translation and minimization policy is similar to PMON with the following exceptions:

- Observation statements are applied to the connecting state at the next time-point. This is done in order to ensure that what we actually observe is the fixpoint result of the cascade at the present time-point. For example:

  obs $\quad [5]\mathsf{dead} \wedge \neg\mathsf{walking}$ (4.13)

  would be translated to

  obs $\quad \mathit{Holds}(6,0,\mathsf{dead}) \wedge \neg \mathit{Holds}(6,0,\mathsf{walking})$.

  The same goes for static constraints and preconditions in actions.

- All reassignments take effect in the primary-result state according to translation (4.9).

Let Υ be a action scenario, and let $Tran(\Upsilon)$ be its translation into $\mathcal{L}$(FL). Then the formula α is entailed by $Tran(\Upsilon)$ iff

$$\Gamma_{UNA} \wedge \Gamma_T \wedge \Gamma_{NCG2} \wedge \Gamma_{CP} \wedge \Gamma_{OBS} \wedge \Gamma_{ACC} \wedge$$
$$Circ_{SO}(\Gamma_{SCD} \wedge \Gamma_{DEP}; Occlude) \models \alpha \qquad (4.14)$$

Note that the the *Occlude* predicate only occurs positively in a translation of an action scenario. Consequently, we can show that the circumscription above is reducible to a logically equivalent first-order formula.

Example 4.5.1
This example of a simple seesaw shows the translation of a scenario from the surface language to $\mathcal{L}$(FL). The example consists of two different actions influencing a seesaw, MoveLeft and MoveRight, each having the position as argument. The actions are used to move one of the sides of the seesaw either up or down; the idea is that the other side should move in the opposite direction without any intermediate states that violate the rule that the seesaw is straight. Furthermore, note that this is a theory with causal cycles containing four dependency laws governing the behavior of the seesaw.

Scenario Description 4.15
obs_1 $[0]\mathsf{left} \mathrel{\hat{=}} \mathsf{down} \wedge \mathsf{right} \mathrel{\hat{=}} \mathsf{up}$
occ_1 $[2,3]$ MoveRight(down)
occ_2 $[3,4]$ MoveLeft(down)
occ_3 $[5,7]$ MoveLeft(up)
acs_1 $[t_1,t_2]$ MoveRight($position$) $\rightsquigarrow$ $[t_1,t_2]$ right $:=$ $position$
acs_2 $[t_1,t_2]$ MoveLeft($position$) $\rightsquigarrow$ $[t_1,t_2]$ left $:=$ $position$
dep_1 $\forall t([t]\mathsf{left} \mathrel{\hat{=}} \mathsf{down} \gg [t]\mathsf{right} \mathrel{\hat{=}} \mathsf{up})$
dep_2 $\forall t([t]\mathsf{left} \mathrel{\hat{=}} \mathsf{up} \gg [t]\mathsf{right} \mathrel{\hat{=}} \mathsf{down})$
dep_3 $\forall t([t]\mathsf{right} \mathrel{\hat{=}} \mathsf{down} \gg [t]\mathsf{left} \mathrel{\hat{=}} \mathsf{up})$
dep_4 $\forall t([t]\mathsf{right} \mathrel{\hat{=}} \mathsf{up} \gg [t]\mathsf{left} \mathrel{\hat{=}} \mathsf{down})$

After expansion we get the following scenario description:

Scenario Description 4.16
obs_1 $[0]\mathsf{left} \mathrel{\hat{=}} \mathsf{down} \wedge \mathsf{right} \mathrel{\hat{=}} \mathsf{up}$
scd_1 $[2,3]\mathsf{right} := \mathsf{down}$
scd_1 $[3,4]\mathsf{left} := \mathsf{down}$
scd_1 $[5,7]\mathsf{left} := \mathsf{up}$

dep_1 $\forall t([t]\mathsf{left} \hat{=} \mathsf{down} \gg [t]\mathsf{right} \hat{=} \mathsf{up})$
dep_2 $\forall t([t]\mathsf{left} \hat{=} \mathsf{up} \gg [t]\mathsf{right} \hat{=} \mathsf{down})$
dep_3 $\forall t([t]\mathsf{right} \hat{=} \mathsf{down} \gg [t]\mathsf{left} \hat{=} \mathsf{up})$
dep_4 $\forall t([t]\mathsf{right} \hat{=} \mathsf{up} \gg [t]\mathsf{left} \hat{=} \mathsf{down})$

The corresponding set of labeled well formed formulas in $\mathcal{L}(\mathrm{FL})$ is:

Scenario Description 4.17

obs_1 $Holds(1, 0, \mathsf{left}, \mathsf{down}) \wedge Holds(1, 0, \mathsf{right}, \mathsf{up})$
scd_1 $Holds(3, 1, \mathsf{right}, \mathsf{down}) \wedge Occlude(3, 1, \mathsf{right})$
scd_2 $Holds(4, 1, \mathsf{left}, \mathsf{down}) \wedge Occlude(4, 1, \mathsf{left})$
scd_3 $Holds(7, 1, \mathsf{left}, \mathsf{up}) \wedge \forall t(5 < t \leq 7 \rightarrow Occlude(t, 1, \mathsf{left}))$
dep_1 $\forall t[(Holds(t, 0, \mathsf{left}, \mathsf{down}) \rightarrow Holds(t, 0, \mathsf{right}, \mathsf{up})) \wedge$
$\forall s \geq 0[\neg Holds(t, s, \mathsf{left}, \mathsf{down}) \wedge Holds(t, s+1, \mathsf{left}, \mathsf{down}) \rightarrow$
$(Holds(t, s+2, \mathsf{right}, \mathsf{up}) \wedge Occlude(t, s+2, \mathsf{right}))]]$
dep_2 $\forall t[(Holds(t, 0, \mathsf{left}, \mathsf{up}) \rightarrow Holds(t, 0, \mathsf{right}, \mathsf{down})) \wedge$
$\forall s \geq 0[\neg Holds(t, s, \mathsf{left}, \mathsf{up}) \wedge Holds(t, s+1, \mathsf{left}, \mathsf{up}) \rightarrow$
$(Holds(t, s+2, \mathsf{right}, \mathsf{down}) \wedge Occlude(t, s+2, \mathsf{right}))]]$
dep_3 $\forall t[(Holds(t, 0, \mathsf{right}, \mathsf{down}) \rightarrow Holds(t, 0, \mathsf{left}, \mathsf{up})) \wedge$
$\forall s \geq 0[\neg Holds(t, s, \mathsf{right}, \mathsf{down}) \wedge Holds(t, s+1, \mathsf{right}, \mathsf{down}) \rightarrow$
$(Holds(t, s+2, \mathsf{left}, \mathsf{up}) \wedge Occlude(t, s+2, \mathsf{left}))]]$
dep_4 $\forall t[(Holds(t, 0, \mathsf{right}, \mathsf{up}) \rightarrow Holds(t, 0, \mathsf{left}, \mathsf{down})) \wedge$
$\forall s \geq 0[\neg Holds(t, s, \mathsf{right}, \mathsf{up}) \wedge Holds(t, s+1, \mathsf{right}, \mathsf{up}) \rightarrow$
$(Holds(t, s+2, \mathsf{left}, \mathsf{down}) \wedge Occlude(t, s+2, \mathsf{left}))]]$

Assuming the domain of left and right is $\{\mathsf{down}, \mathsf{up}\}$, circumscription of Γ_{SCD} and Γ_{DEP} with *Occlude* minimized results in the following definition:

$$
\begin{aligned}
\forall t, s, f \qquad & Occlude(t, s, f) \leftrightarrow \\
& [(t = 3 \wedge s = 1 \wedge f = \mathsf{right}) \vee \\
& (t = 4 \wedge s = 1 \wedge f = \mathsf{left}) \vee \\
& (5 < t \leq 7 \wedge s = 1 \wedge f = \mathsf{left}) \vee \\
& (f = \mathsf{right} \wedge \forall t, s_2, v(s = s_2 + 2 \rightarrow \\
& \qquad \neg Holds(t, s_2, \mathsf{left}, v) \oplus Holds(t, s_2 + 1, \mathsf{left}, v))) \vee \\
& (f = \mathsf{left} \wedge \forall t, s_2, v(s = s_2 + 2 \rightarrow \\
& \qquad \neg Holds(t, s_2, \mathsf{right}, v) \oplus Holds(t, s_2 + 1, \mathsf{right}, v)))]
\end{aligned}
$$

The two last lines state that if the value of left is changed, then right should be occluded and vice versa. The same scenario would in TAL result in that the

position of the seesaw would vary freely. For example, it would be impossible to deduce the position of the seesaw at time-point 1 using TAL. □

Example 4.5.2
In the following example we will consider two-way directionality combined with nondeterminism. Let us consider the domain constraint $\mathsf{Sw}_1 \vee \mathsf{Sw}_2 \leftrightarrow \mathsf{La}$ representing that a lamp (La) is on if any of the two switches (Sw_1 and Sw_2) is on. Suppose we can influence both the switches and the lamp via actions. What happens if we turn on a switch? We do not want the other switch to change as well. How can we recognize whether the lamp is influenced directly (leading to possible change in the switches) or if it is influenced via the domain constraint in a left to right direction (not leading to a right to left direction)?

We model the problem as follows:

$$\forall t \geq 0([t]\neg\mathsf{La} \rightarrow ([t](\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \gg [t]\mathsf{La}))$$
$$\forall t \geq 0([t]\mathsf{La} \rightarrow ([t]\neg(\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \gg [t]\neg\mathsf{La}))$$
$$\forall t \geq 0([t](\neg\mathsf{Sw}_1 \wedge \neg\mathsf{Sw}_2) \rightarrow ([t]\mathsf{La} \gg [t](\mathsf{Sw}_1 \vee \mathsf{Sw}_2)))$$
$$\forall t \geq 0([t](\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \rightarrow ([t]\neg\mathsf{La} \gg [t]\neg(\mathsf{Sw}_1 \vee \mathsf{Sw}_2)))$$

This formalization may appear to be very complex, even contradictory. How can one have the precondition La and the effect $\neg\mathsf{La}$, at the same time-point? The answer is that the precondition is checked earlier in the cascade than the consequence occurs. The first dependency law should be read "If the lamp is off and any of the switches are turned on at time-point t, then the result at the end of the cascade at time-point t is that the lamp is on".[7]

Since all of these are influencing constraints with $t = \tau_1(t) = \tau_2(t)$, this

[7]It can be argued that the natural thing to do is to minimize change as much as possible. If we want to do this, we can rewrite the third dependency law as follows:

$$\forall t \geq 0([t](\neg\mathsf{Sw}_1 \wedge \neg\mathsf{Sw}_2) \rightarrow ([t]\mathsf{La} \gg [t]((\mathsf{Sw}_1 \wedge \neg\mathsf{Sw}_2) \vee (\neg\mathsf{Sw}_1 \wedge \mathsf{Sw}_2))))$$

is transformed (and abbreviated) into:

$$
\begin{aligned}
&\forall t \geq 0(\{t,0\}(\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \leftrightarrow \{t,0\}\mathsf{La}) \wedge \\
&\forall t \geq 0, s \geq 0((\{t,s\}\neg(\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \wedge \{t,s+1\}(\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \wedge \\
&\qquad\qquad \{t,s+1\}\neg\mathsf{La}) \rightarrow \\
&\qquad (\{t,s+2\}\mathsf{La} \wedge \{t,s+2\}\mathcal{X}(\mathsf{La}))) \wedge \\
&\forall t \geq 0, s \geq 0((\{t,s\}(\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \wedge \{t,s+1\}\neg(\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \wedge \\
&\qquad\qquad \{t,s+1\}\mathsf{La}) \rightarrow \\
&\qquad (\{t,s+2\}\neg\mathsf{La} \wedge \{t,s+2\}\mathcal{X}(\mathsf{La}))) \wedge \\
&\forall t \geq 0, s \geq 0((\{t,s\}\neg\mathsf{La} \wedge \{t,s+1\}\mathsf{La} \wedge \{t,s+1\}(\neg\mathsf{Sw}_1 \wedge \neg\mathsf{Sw}_2)) \rightarrow \\
&\qquad (\{t,s+2\}(\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \wedge \\
&\qquad\ \{t,s+2\}(\mathcal{X}(\mathsf{Sw}_1) \wedge \mathcal{X}(\mathsf{Sw}_2)))) \wedge \\
&\forall t \geq 0, s \geq 0((\{t,s\}\mathsf{La} \wedge \{t,s+1\}\neg\mathsf{La} \wedge \{t,s+1\}(\mathsf{Sw}_1 \vee \mathsf{Sw}_2)) \rightarrow \\
&\qquad (\{t,s+2\}\neg(\mathsf{Sw}_1 \vee \mathsf{Sw}_2) \wedge \\
&\qquad\ \{t,s+2\}(\mathcal{X}(\mathsf{Sw}_1) \wedge \mathcal{X}(\mathsf{Sw}_2))))
\end{aligned}
$$

The length of this result may be surprising, but this example contains many subtleties that are not obvious at first glance. For example, toggling on just one switch should not lead to a model where both switches are on. □

The next section explores a completely different way of approaching the problem with cycles, without any of the modifications of TAL necessary in this section.

4.6 Breaking Causal Cycles: II

This section consists of an unpublished paper by Gustafsson and Kvarnström that discusses a solution to the problems associated with causal cycles in the context of TAL-C. Although TAL-C has not yet been presented (see the next chapter or Appendix C) we have chosen to place this presentation here since the aim is to provide a concise solution to the problem with causal cycles. The readers not familiar with TAL-C are encouraged to read Chapter 5 before reading the following section.

4.6.1 Introduction

There are a number of approaches in the literature that are able to handle causal cycles. Interestingly, it appears that these approaches always use non-

standard entailment criteria specifically designed to deal with the ramification problem (for example, Thielscher [98], McCain and Turner [71], and Denecker et al. [17]), while approaches where an existing logic is extended for side effects have problems with causal cycles, allowing them to trigger themselves (e.g., Lin [67], Gustafsson and Doherty [42], and Shanahan [94]).

This immediately leads to the question of whether using a tailor-made entailment criterion is truly essential for dealing with causal cycles. This would be very unfortunate, as it would preclude the use of many standard techniques and tools such as first order theorem provers. As we will show, this is not the case. We will present an existing solution to the ramification problem within the TAL (Temporal Action Logics) framework [42] and demonstrate that it allows spontaneous change when causal cycles are present. We will then describe a filtering technique for removing all models in which such spontaneous change is present. This is achieved by introducing additional logical constraints on causal effects, and is handled completely within the logic without recourse to tailor-made entailment criteria. Therefore, the resulting TAL narratives are still reducible to first-order logic, existing correctness results are still applicable, and existing solutions to problems such as the qualification problem can still be used.

4.6.2 Self-triggered Cycles in TAL-C

To see how causal cycles can be triggered spontaneously in TAL-C, consider the following narrative, inspired by an example by Denecker et al. [17].

Example 4.6.1
There are two interlocking gear wheels. If either wheel is turning, this will cause the other wheel to turn. Initially the wheels are still, but between time-points 4 and 5, an action makes the first wheel start turning.

Scenario Description 4.18

per	$Per(\mathsf{T}_1) \wedge Per(\mathsf{T}_2)$
acs	$[t_1, t_2]\,\mathsf{Turn}_1 \rightarrow I((t_1, t_2]\,\mathsf{T}_1)$
dep_1	$\forall t.\, C_T([t]\,\mathsf{T}_1) \rightarrow I([t]\,\mathsf{T}_2)$
dep_2	$\forall t.\, C_T([t]\,\mathsf{T}_2) \rightarrow I([t]\,\mathsf{T}_1)$
obs	$[0]\,\neg\mathsf{T}_1 \wedge \neg\mathsf{T}_2$
occ	$[4, 5]\,\mathsf{Turn}_1$

We declare the fluents T_1 and T_2 as persistent, and describe the effects of the action Turn_1. The two dependency constraints describe the fact that if either wheel is turning, then this is sufficient cause for the other wheel to

turn. Finally, we observe that neither wheel is turning at 0, and an action occurrence statement states that Turn_1 takes place between 4 and 5.

Intuitively, one would expect both wheels to remain still from 0 to 4. At 5, we turn T_1, which should also cause T_2 to start turning. Although there is a model where this happens, there are also models in which T_1 starts turning at some time-point $t < 5$. This is sufficient cause for T_2 to start turning, which is sufficient cause for T_1 to start turning. The causal cycle has triggered itself spontaneously. □

4.6.3 Preventing Self-triggered Cycles

Our approach to filtering models with self-triggered causal cycles will be presented in two steps. First, we will present a method that handles causal dependencies triggered by a change in a single fluent. We will then extend this method to handle dependencies triggered by changes in multiple fluents as well as dependencies triggered by fluents taking on certain values rather than changing values.

Simple Trigger Conditions

We first consider action definitions and dependency constraints of the form

$$\begin{array}{ll} \mathsf{acs} & [t_1, t_2]\,\mathsf{Action} \rightarrow (\phi \rightarrow I((\tau, \tau']\,\psi)) \text{ and} \\ \mathsf{dep} & \forall t. C_T([t]\, f \mathrel{\hat{=}} v) \rightarrow I([t+c]\,\psi) \end{array} \tag{4.19}$$

where $c \in \{0, 1, 2, \ldots\}$. If $c = 0$, dep is a *non-delayed dependency constraint*; otherwise, it is a *delayed dependency constraint.*

Returning briefly to the gear example, note that there are at least two different ways of defining a causal cycle. Consider a spurious model where dep_1 and dep_2 self-trigger at time 1. It could be said that there is a causal cycle involving dep_1 and dep_2, since dep_1 triggered dep_2 which in turn triggered dep_1. Alternatively, it could be said that there is a causal cycle involving the fluents T_1 and T_2, since a change in T_1 caused T_2 to change, which caused the initial change in T_1. We will use the latter approach: Given a model of a TAL narrative and a time-point τ, we define a *triggered causal cycle* as a sequence $\langle f_1, \ldots, f_m, f_{m+1} \rangle$, where $f_{m+1} = f_1$ and where for all $1 \leq k \leq m$, there is a non-delayed dependency constraint $\forall t. C_T([t]\, f_k \mathrel{\hat{=}} v) \rightarrow I([t]\,\psi)$ such that $C_T([\tau]\, f_k \mathrel{\hat{=}} v)$ holds and such that f_{k+1} occurs in ψ. Such cycles can be detected using a *cause graph.*

Definition 4.6.1 (Cause Graph)
Let Υ be a TAL narrative and $\mathcal{F}$ the set of fluents in Υ. Let grounded be a symbol representing a grounded cause, and let $\mathcal{F}' = \mathcal{F} \cup \{\mathsf{grounded}\}$. The unique *cause graph* for a time-point τ and a model of Υ is a directed graph with nodes $\mathcal{F}'$, constructed as follows. First, for each action a occurring in Υ and for each fluent f affected by a at τ, add an edge from grounded to f. Then, for each dependency constraint $\forall t. C_T([t]\, g \hat{=} v) \rightarrow I([t{+}c]\, \psi)$ triggered at $t = \tau - c$ and for each fluent f occurring in ψ, if $c = 0$ then add an edge from g to f, otherwise add an edge from grounded to f. □

However, simply removing every model for which there is some τ with a cyclic cause graph is too restrictive. In the gear example, an action will make T_1 true at 5; even in the intended model, this causes T_2 to become true at 5, which causes T_1 to become true. Thus, there is a triggered causal cycle, but since it has a grounded cause (the action), this model should be retained.

Instead, a model should be removed when there is some f that has changed values at some τ without a grounded cause, that is, some f for which there is no path from grounded to f in the cause graph for τ. Equivalently, a model should be removed when there is some τ for which it has no acyclic *selected cause graph.*

Definition 4.6.2 (Selected Cause Graph)
Given a time-point τ and a model of a TAL narrative, a *selected cause graph* for this time-point and model is a subgraph of the corresponding cause graph, with the same nodes, such that for each fluent f that has changed at τ, there is at least one edge leading to f (a *reason* for the change). □

Note that a cause graph contains edges to every fluent that has been influenced by an action or dependency constraint, even if some of those fluents have not actually changed values. This provides some useful information when we consider diagnosis. In a *selected* cause graph, however, we only consider fluents that have changed values.

Example 4.6.2 (continued from 4.6.1)
For any model of the gear example, the cause graph for time-point 5 has the nodes T_1, T_2 and grounded, with edges as shown in Figure 4.2. Although this graph is cyclic, the part of the figure marked in gray is an acyclic selected cause graph.
However, in any model where the causal cycle triggers itself at some τ, the cause graph for τ has only two edges (Figure 4.3). This graph is cyclic, and if we remove any edge, it is no longer a selected cause graph. □

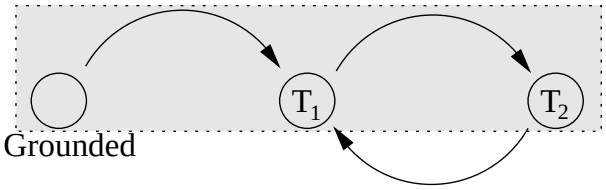

Figure 4.2: Cause graph for time 5

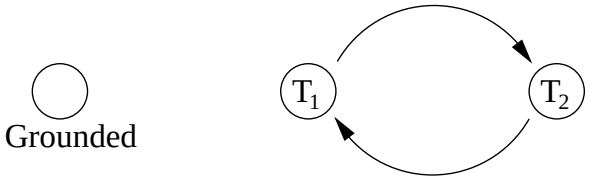

Figure 4.3: Cause graph with self-triggered cycle

We must now define a set of TAL formulas that model the cause graph and the existence of an acyclic selected cause graph at each time-point. We introduce a value domain node containing the special value grounded as well as one symbol for each fluent in the original narrative. The cause graph is modeled as a boolean durational fluent Cause(node, node) with default value false. Then, by introducing additional consequences for each action and dependency constraint, $\mathsf{Cause}(n, n')$ is made true at τ iff there is an edge from n to n' in the cause graph for τ.

For each action definition of the form $[t_1, t_2]\,\mathsf{Action} \rightarrow (\phi \rightarrow I((\tau, \tau']\,\psi))$, let the set $\{n_1, \ldots, n_m\} \subseteq \mathsf{node}$ be those nodes that correspond to some fluent occurring in ψ, and replace the action definition with $[t_1, t_2]\,\mathsf{Action} \rightarrow (\phi \rightarrow I((\tau, \tau']\,\psi) \wedge \bigwedge_{i=1}^{m} I((\tau, \tau']\,\mathsf{Cause}(\mathsf{grounded}, n_i)))$, indicating that there is a grounded cause for change in each n_i.

For each dependency constraint $\forall t. C_T([t]\, f \hat{=} v) \rightarrow I([t + c]\,\psi)$, let the set $\{n_1, \ldots, n_m\} \subseteq \mathsf{node}$ be those nodes that correspond to some fluent in ψ, and replace the dependency constraint with $\forall t. C_T([t]\, f \hat{=} v) \rightarrow I([t + c]\,\psi) \wedge \bigwedge_{i=1}^{m} I([t + c]\,\mathsf{Cause}(n, n_i))$, where n is the node corresponding to f if $c = 0$ and $n = \mathsf{grounded}$ otherwise.

Given these modified statements, $\mathsf{Cause}(n, n')$ will be true at any given time-point τ iff the cause graph for τ has an edge from n to n'. (Recall that we circumscribe *Occlude* in the $\mathcal{L}$(FL) translation of the narrative. Since Cause is a durational fluent, taking on the value false unless occluded, we have essentially also circumscribed Cause. This explains why Cause is never allowed to vary freely.)

A boolean dynamic fluent Selected(node, node) is added, together with domain constraints forcing it to correspond to some selected cause graph. For each fluent f with corresponding node n, a domain constraint of the following form is added:

$$\mathsf{acc}_{1a} \qquad ([t]\, f \mathrel{\hat{=}} v \not\equiv [t+1]\, f \mathrel{\hat{=}} v) \rightarrow \exists c[[t+1]\, \mathsf{Selected}(c, n)] \tag{4.20}$$

Then, we ensure that Selected is a subgraph of Cause:

$$\mathsf{acc}_{1b} \qquad [t]\, \mathsf{Selected}(n, n') \rightarrow [t]\, \mathsf{Cause}(n, n') \tag{4.21}$$

Now, Selected must be *some* selected cause graph. In order to ensure that there exists at least one *acyclic* selected cause graph, all that remains is to constrain Selected to be acyclic, which is the case iff its transitive closure is irreflexive. Although the transitive closure cannot be defined in first-order logic, it is irreflexive iff it and all its supersets are irreflexive. Therefore, we define a new boolean dynamic fluent Closure, which is constrained to be some irreflexive superset of the transitive closure of Selected.

$$\mathsf{acc}_{1c} \qquad [t]\, \mathsf{Selected}(n, n') \rightarrow \mathsf{Closure}(n, n') \tag{4.22}$$

$$\mathsf{acc}_{1d} \qquad [t]\, \mathsf{Selected}(n, n') \wedge \mathsf{Closure}(n', n'') \rightarrow \mathsf{Closure}(n, n'')$$

$$\mathsf{acc}_{1e} \qquad [t] \neg \mathsf{Closure}(n, n)$$

We emphasize that each step in this procedure can be performed mechanically by a narrative pre-processor. This removes all models in which a causal cycle has triggered itself spontaneously.

Complex Trigger Conditions

The solution presented above was sufficient for handling dependency constraints triggered by a change in a single fluent. We would also like to handle dependency constraints triggered by changes in multiple fluents, as well as by fluents that *have* a certain value rather than *change* values. We will therefore consider constraints of the form

$$\mathsf{dep} \quad \forall t.(\textstyle\bigwedge_{i=1}^{n} [t + c_i]\, f_i \mathrel{\hat{=}} v_i) \rightarrow I([t + c]\, \psi), \tag{4.23}$$

where $c, c_i \in \{0, 1, 2, \ldots\}$. Note that although the C_T macro is no longer allowed, any formula allowed in the previous section can easily be rewritten on this form. Also, dependency constraints with disjunctive preconditions can be written as multiple dependency constraints on this form. Disjunctive effects are allowed within the scope of the I macro operator.

We must now change the definitions of cause graphs and selected cause graphs, since no *single* fluent can be selected as the cause of change in another fluent: It is necessary to select the entire *set* of fluents that were the trigger condition for some dependency constraint.

Definition 4.6.3 (Cause Graph)
Let Υ be a TAL narrative and $\mathcal{F}$ the set of fluents in Υ. Let grounded be a symbol representing a grounded cause, and let $\mathcal{F}' = \mathcal{F} \cup \{\mathsf{grounded}\}$. The unique *cause graph* for a time-point τ and a model of Υ is a labeled directed multigraph with nodes $\mathcal{F}'$, constructed as follows. First, for each action a occurring in Υ and for each fluent f affected by a at τ, add an edge from grounded to f with the label $\langle a, \mathsf{true}\rangle$. Then, for each dependency constraint $dep = \forall t.(\bigwedge_{i=1}^{n}[t + c_i]\, f_i \mathrel{\hat{=}} v_i) \rightarrow I([t + c]\, \psi)$ triggered at some $t = \tau - c$, for each $1 \leq i \leq n$, and for each fluent f occurring in ψ, if $c_i < c$ (a local delay) add an edge from f_i to f labeled $\langle dep, \mathsf{true}\rangle$, otherwise add an edge from f_i to f labeled $\langle dep, \mathsf{false}\rangle$. □

Example 4.6.3 (continued)
We rewrite the dependency constraints on the required form:

Scenario Description 4.24
dep_1 $\forall t.([t]\, \neg \mathsf{T}_1 \wedge [t+1]\, \mathsf{T}_1) \rightarrow I([t+1]\, \mathsf{T}_2)$
dep_2 $\forall t.([t]\, \neg \mathsf{T}_2 \wedge [t+1]\, \mathsf{T}_2) \rightarrow I([t+1]\, \mathsf{T}_1)$

The cause graph for time-point 5 is now somewhat more complex (Figure 4.4). Each label is a tuple $\langle s, g\rangle$, where s is the name of the action or dependency constraint that introduced the edge and g indicates whether the edge is grounded. □

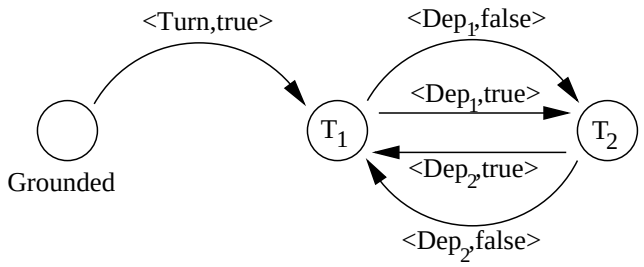

Figure 4.4: Cause graph for time 5

The condition for filtering out models must change slightly, since delayed dependencies, which previously gave rise to edges from the special value grounded to the fluents they affected, now introduce edges from the fluents in their preconditions to the fluents they affect. A model should be removed iff there is some time-point τ for which every selected cause graph for the

model contains some cycle where every edge has a label $\langle l_i, \mathsf{false}\rangle$, that is, where every edge arises from a non-delayed fluent dependency.

Definition 4.6.4 (Selected Cause Graph)
Given a time-point τ and a model of a TAL narrative, a *selected cause graph* S for this time-point and model is a subgraph of the corresponding cause graph C, with the same nodes, such that for each fluent f that has changed from $\tau - 1$ to τ, there is at least one edge leading to f, and for each edge label l in C, either all edges in C labeled l are in S, or S has no edge labeled l. □

As before, we introduce a TAL value domain node containing the special value grounded as well as one value for each fluent in the original TAL narrative. We also introduce a value domain label containing one symbol for each action and dependency constraint, as well as a special symbol indirect which will be used later. Then, the cause graph is modeled as a boolean durational fluent $\mathsf{Cause}(\mathsf{node}, \mathsf{node}, \mathsf{label}, \mathsf{boolean})$ with default value false, and $\mathsf{Cause}(n, n', l, g)$ is made true whenever the cause graph has an edge from n to n' labeled $\langle l, g\rangle$. This is achieved by introducing additional consequences for each action and dependency constraint.

For each action definition of the form $[t_1, t_2]\,\mathsf{Action} \rightarrow (\phi \rightarrow I((\tau, \tau']\,\psi))$, let $\{n_1, \ldots, n_m\} \subseteq \mathsf{node}$ be those nodes that correspond to some fluent in ψ and let $\mathsf{a} \in \mathsf{label}$ be the identifier for this action. Replace the definition with $[t_1, t_2]\,\mathsf{Action} \rightarrow (\phi \rightarrow I((\tau, \tau']\,\psi) \wedge \bigwedge_{i=1}^{m} I((\tau, \tau']\,\mathsf{Cause}(\mathsf{grounded}, n_i, \mathsf{a}, \mathsf{true})))$.

Then, for each dependency constraint of the form $\forall t.(\bigwedge_{i=1}^{n}[t + c_i]\,f_i \mathrel{\hat{=}} v_i) \rightarrow I([t + c]\,\psi)$ with identifier $\mathsf{d} \in \mathsf{label}$, let $\langle p_1, \ldots, p_n\rangle$ be the nodes corresponding to $\langle f_1, \ldots, f_n\rangle$ and let $\{n_1, \ldots, n_m\} \subseteq \mathsf{node}$ be those nodes that correspond to some fluent in ψ. For all $1 \leq i \leq n$, let g_i be the value true if $c < c_i$, the value false otherwise. Replace the dependency constraint with $\forall t.(\bigwedge_{i=1}^{n}[t+c_i]\,f_i \mathrel{\hat{=}} v_i) \rightarrow I([t+c]\,\psi) \wedge \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{m} I([t+c]\,\mathsf{Cause}(p_i, n_j, \mathsf{d}, g_i))$.

A boolean dynamic fluent Selected with the same arguments as Cause is added, as well as domain constraints forcing it to correspond to some selected cause graph. For each fluent f with corresponding node n, a domain constraint is added:

$$\mathsf{acc}_{2a} \qquad ([t-1]\,f \mathrel{\hat{=}} v \not\equiv [t]\,f \mathrel{\hat{=}} v) \rightarrow \exists r, c, g[[t]\,\mathsf{Cause}(c, f, r, g) \wedge \forall c'[[t]\,\mathsf{Cause}(c', f, r, g) \rightarrow [t]\,\mathsf{Selected}(c', f, r, g)]] \tag{4.25}$$

Then, we ensure that Selected is a subgraph of Cause:

$$\mathsf{acc}_{2b} \qquad [t]\,\mathsf{Selected}(a, b, r, g) \rightarrow [t]\,\mathsf{Cause}(a, b, r, g) \tag{4.26}$$

Finally, we should remove all models for which there is some time-point for which it is impossible to find a definition of Selected that satisfies the conditions given above.

$$\mathsf{acc}_{2c} \quad [t]\,\mathsf{Selected}(n, n', r, \mathsf{false}) \rightarrow \mathsf{Closure}(n, n') \tag{4.27}$$
$$\mathsf{acc}_{2d} \quad [t]\,\mathsf{Selected}(n, n', r, \mathsf{false}) \wedge \mathsf{Closure}(n', n'') \rightarrow \mathsf{Closure}(n, n'')$$
$$\mathsf{acc}_{2e} \quad [t]\,\neg\mathsf{Closure}(n, n)$$

4.6.4 Diagnosis

The techniques presented in the previous section reified dependencies between fluents to prevent causal cycles from being triggered spontaneously. One of the advantages of having reified this information is that it can also be used for other purposes, such as diagnosis.

Suppose, for example, that we want to know why Tweety has died. More formally, alive has become false at some time-point t, and we want to find the underlying causes. This information is encoded in the selected cause graphs for t. Each graph corresponds to one combination of direct and indirect effects that caused Tweety to die.

Given a selected cause graph S, consider every edge leading to the node alive. Each such edge has influenced the fluent directly in some way: If it is labeled $\langle A_i, \mathsf{true}\rangle$ then the action A_i has changed the value of alive. If it is labeled $\langle \mathsf{dep}_i, \mathsf{true}\rangle$, then the delayed dependency constraint dep_i has affected the fluent. Otherwise, it is labeled $\langle \mathsf{dep}_i, \mathsf{false}\rangle$, and the non-delayed dependency constraint dep_i has affected the fluent; in this case, it is possible to use the selected cause graph to track the reasons why dep_i was triggered, and to recursively follow these reasons back to their grounded causes.

4.6.5 Related Work

We will now select four other solutions and discuss their relation to our approach with respect to causal cycles.

Thielscher. One of the earliest causality-based solutions to the ramification problem is Thielscher's approach [98, 99], which uses a specialized entailment criterion where applying an action to a state leads to a new intermediate state. If at least one causal rule can be applied in this state, one such rule is chosen nondeterministically, which leads to a new intermediate state. This is repeated until a stable state is reached. Each fluent may change values more than one time in this sequence of states, which suggests that there is an actual flow of time within the sequence.

Since causal rules never have direct or indirect effects in the state in which they are triggered, they can clearly never be self-triggered. Unfortunately, the procedural nature of this approach makes it hard to compare it to our more declarative approach, and although there has been much debate about whether this is the best way to model causal delays, such discussions are outside the scope of this thesis.

Nondeterminism is attained through the freedom to choose in what order the causal rules should be applied. This suggests that it might be hard to foresee the outcome of a complex system where the intermediate states have no correspondence to reality. A more thorough investigation of Thielscher's approach can be found in Thielscher [99] and Denecker et al. [17].

Denecker, Dupré and Belleghem. Denecker et al. [17] present a general semantics for handling causal rules, using an inductive definition as the basis for the reasoning mechanism. They point out that it is necessary to restrict causal cycles on a semantical level (as in the approach presented in this section) rather than on a syntactical level.

A central concept is the $\mathcal{D}$-proof-tree, which is used to define the set of legal models and has been the main source of inspiration for our approach. A graph of dependencies is built for each fluent, and if that graph is a tree, it determines the value of the fluent.

Denecker et al. make a strong differentiation between state constraints and causal rules, and claim that it should be possible to express causal rules without an underlying state constraint. This differentiation is not needed in TAL, however, since TAL dependency constraints are flexible enough to express both kinds of rules.

Another important difference is that while [17] represents causality in terms of a *Cause* predicate, which can be used in the trigger of a causal rule, TAL dependency constraints are triggered by fluent values. Problematic statements like $(cause(p) \leftarrow \neg cause(q)) \wedge (cause(q) \leftarrow \neg cause(p))$ are not expressible in TAL.

Shanahan. Shanahan [94] notes in his solution to the ramification problem in the event calculus that mutually dependent fluents can cause phantom self-starting events. However, his approach contains most of the requirements for the methods presented in this section to be applicable. Cause is similar to Shanahan's *Happens* predicate in that it is false by default, and dynamic fluents such as Selected and Closure can easily be handled by adding a *Releases* predicate as described in Shanahan [93].

Lin. Lin [67] was the first to consider the problem of causal cycles in relation to causal domain constraints. Rather than preventing causal cycles from triggering themselves spontaneously, he defines a syntactical *stratification* condition that ensures that a theory cannot contain causal cycles at all. This condition is stronger than the semantic counterpart of [17].

It is doubtful whether one could remove the stratification condition from Lin's approach by applying the techniques presented here, since stratification is also (according to Lin) necessary because the Clark completion being used is too weak for handling cycles and recursion in causal rules.

4.7 Conclusions

We feel that PMON and its extension TAL have a number of advantages over other formalisms for specifying action scenarios. They use explicit time in terms of a linear metric time structure which allows one to specify actions with duration, delayed effects of actions, and the incomplete specification of timing and order of actions in a straightforward manner. We have argued that there is a simple and intuitive surface language for describing scenarios and that for this particular class of scenario descriptions, we can reduce the circumscribed scenarios to the first-order case algorithmically.

On the other hand, we have not claimed that this logic is suitable for all classes of problems, nor that it has solved the frame and ramification problems in total, whatever they may be defined as being for the moment.

We have shown that in the standard approach to ramification in TAL, causal cycles can be triggered spontaneously. Two solutions to this are described and discussed. The first utilizes a tailor-made entailment criterion with sequence of intermediate states similar to the approaches by Thielscher and Sandewall.

We posed the question of whether correcting the problem with cycles always requires this type tailor-made entailment criterion, and showed that this was not the case by providing a systematic method for adding constraints to a narrative in order to filter out unwanted models. This method can be completely automated and implemented as a simple narrative preprocessor. Since the TAL-C language and its underlying logic and entailment criterion needed no modifications, any narrative would still be reducible to first-order logic, and existing solutions to other problems such as the qualification problem could still be used.

We also showed that our approach can help in performing some forms of diagnosis and explanation, and compared it to several other approaches in the literature.

The next chapter will illustrate how dependency rules can be used to solve several aspects of the concurrency problem.

Chapter 5

Reasoning about Concurrent Interaction

To an agent operating in a complex multi-agent environment, it is important to be able to reason about how the environment changes due to the actions that the agent performs. Most of the work in reasoning about action and change has been done under the (sometimes implicit) assumption that there is a single agent that performs sequences of actions. This is a comparatively easier modeling problem than reasoning about action and change under a multi-agent assumption, or under a non-sequentiality assumption for a single agent, which by necessity introduces the complication that one or several agents can perform actions concurrently. Concurrency, in turn, can involve a wide range of interactions between actions, which makes it unlikely that there is one single, uniform technique of general applicability.

The work presented here is based on the article "Concurrent Interaction" by Karlsson and Gustafsson [48].

5.1 TAL-C

This chapter presents an approach to reasoning about action and change which supports the description of concurrent actions with nontrivial interactions. The approach is based on TAL 1.0 (*Temporal Action Logic*, called PMON and PMON(RCs) in the previous chapters), which in its original form [90, 20] covers worlds with natural numbers time domain and sequentially executed actions whose effects can be context-dependent (different initial states can lead to different effects) and nondeterministic (the same initial state can lead to several different effects). From the perspective of concurrency, an im-

portant property of TAL is that actions have durations (that is, occur over an interval of time). In the previous chapter the original PMON logic was extended to support ramifications, or indirect effects [42]. In this chapter we present TAL-C, a development of the TAL/PMON formalism, which combines support for ramification and concurrency. Some of the results presented in this chapter have later been exploited for dealing with delayed effects [49] which is described in Chapter 6, and for handling qualified action descriptions [52].

In TAL-C, the description of concurrent interactions is made on the level of features (state variables). The central idea is that actions are not modeled to directly change the world state, but to produce influences on features. For each feature one can then use influence laws to specify how it is affected by its various associated influences. Besides providing a flexible and versatile means for describing concurrent interactions, the use of influences and influence laws permits describing the properties of actions, dependencies and features in isolation, thereby improving modularity. The major merit of TAL-C is that it combines (a) a standard first-order semantics and proof theory; (b) a notion of explicit time, which makes it possible to reason about the durations and timing of actions and effects; and (c) the modeling of a number of important phenomena related to concurrency, in addition to ramification and nondeterminism.

5.1.1 The Two Language Levels of TAL-C

TAL-C, like its predecessors, is a formalism consisting of two languages. First, there is the surface language $\mathcal{L}$(ND) which provides a number of macros that make it possible to describe TAL-C scenarios in a concise way. Scenario descriptions in $\mathcal{L}$(ND) are translated to the standard first-order language $\mathcal{L}$(FL) by expanding these macros, and after a suitable circumscription policy has been applied, and reduced to a first-order formula, first-order deduction can then be used for reasoning about the scenario descriptions. $\mathcal{L}$(ND) is used throughout most of this thesis for readability; the translation to $\mathcal{L}$(FL) is presented in Appendix C. Therefore, some definitions are given preliminary presentations in $\mathcal{L}$(ND) but are actually encoded as macro expansions in the translation to $\mathcal{L}$(FL).

5.1.2 Organization of the Chapter

The rest of the chapter is organized as follows. In Section 5.2, TAL is introduced with an example and a definition of the syntax of the surface language

$\mathcal{L}$(ND). In Section 5.3, a number of interesting cases of concurrency are identified and discussed. Section 5.4 outlines an approach to concurrency and introduces the concept of influences, and Section 5.5 introduces the extensions to TAL that results in TAL-C. In Section 5.6, the means for solving the problems identified in Section 5.3 are developed. Section 5.7 addresses modularity in the context of TAL-C. Section 5.8 provides an overview of previous work on concurrency, and Section 5.9 contains some conclusions. Appendix C in the thesis presents a translation from the surface language $\mathcal{L}$(ND) to the first-order language $\mathcal{L}$(FL).

5.2 Preliminaries

The surface language $\mathcal{L}$(ND) for sequential scenario descriptions is presented in this section, and extensions for concurrency are introduced in Section 5.5.

5.2.1 Scenario Descriptions in TAL

The following is a scenario description in $\mathcal{L}$(ND). It describes a world with two types of actions (LightFire and PourWater), and a number of agents (bill and bob) and other objects (wood1). For notational convenience, all variables appearing free are implicitly universally quantified.

Scenario Description 5.1

acs_1 $[s,t]\mathsf{LightFire}(a,x) \rightarrow ([s]\mathsf{dry}(x) \wedge \mathsf{wood}(x) \rightarrow R((s,t]\mathsf{fire}(x)))$

acs_2 $[s,t]\mathsf{PourWater}(a,x) \rightarrow$
$(R([s,t]\neg\mathsf{dry}(x)) \wedge ([s]\mathsf{fire}(x) \rightarrow R((s,t]\neg\mathsf{fire}(x))))$

obs_1 $[0]\mathsf{dry}(\mathsf{wood1}) \wedge \neg\mathsf{fire}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1})$

occ_1 $[2,5]\mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})$

occ_2 $[6,7]\mathsf{PourWater}(\mathsf{bob},\mathsf{wood1})$

The statements acs_1 and acs_2 are action laws, which describe the effects of specific action types under different conditions. The first action law states that if an agent a lights a fire using some wood x, and if the wood is dry, then the result will be that the wood is on fire. The expression $[s,t]\mathsf{LightFire}(a,x)$ denotes the action in question where $[s,t]$ is its time interval, and $[s]\mathsf{dry}(x) \wedge \mathsf{wood}(x)$ denotes that the features $\mathsf{dry}(x)$ and $\mathsf{wood}(x)$ hold at time-point s. The statement $R((s,t]\mathsf{fire}(x))$ denotes that the feature $\mathsf{fire}(x)$ is reassigned to become true somewhere in the interval $(s,t]$, and in particular that it is true

at the last time-point t of the interval.[1] The second action law states that if somebody pours water on an object, then the object will no longer be dry, and will cease being on fire.

The line labeled obs_1 is an observation statement. It states that the wood (denoted wood1) is dry and not burning at the initial time-point 0. Observations are assumed to be correct, and can refer to arbitrary time-points or intervals; in the latter case, the notation $[\tau, \tau']\phi$ is used (e.g. $[0,6]\mathsf{dry}(\mathsf{wood1})$). The lines occ_1 and occ_2 are occurrence statements. They describe what actions actually occur in a scenario description. A fire is lit by the agent bill during the temporal interval $[2,5]$, and then the agent bob pours water on the wood during the temporal interval $[6,7]$. No actions besides those explicitly appearing in the occurrence statements are assumed to occur in the scenario. By using non-numerical temporal constants, such as $[\mathsf{s}_1, \mathsf{t}_1]\mathsf{LightFire}(\mathsf{bill}, \mathsf{wood1})$, the exact timing of an action can be left unspecified.

It is possible for features to have domains other than boolean truth values. In that case, the notation $f \hat{=} \omega$ is used to state that the feature f has the value ω, like in the statement $[5]\mathsf{traffic\text{-}light} \hat{=} \mathsf{green}$. The same notation is applicable to booleans (but optional), for example $[s]\mathsf{dry}(x) \hat{=} \mathsf{T}$.

The following are two examples that could complement the description of the LightFire action in (5.1). The line labeled dep_1 states that if an object starts burning, then it also starts smoking, and dep_2 states that if an object starts smoking and the damper is closed (or the object is smoking and the damper becomes closed) then the agent's eyes become sore. The C_T operator denotes that the expression inside was false at the previous time-point, if one exists, and has just become true: $C_T([t]\alpha) \stackrel{\text{def}}{=} (\forall t'[t = t'+1 \rightarrow [t']\neg\alpha] \wedge [t]\alpha)$[2].

$$\begin{aligned} &\mathsf{dep}_1 \quad C_T([t]\mathsf{fire}(x)) \rightarrow R([t]\mathsf{smoking}(x)) \\ &\mathsf{dep}_2 \quad C_T([t]\mathsf{smoking}(x) \wedge \neg\mathsf{damper\text{-}open}) \rightarrow R([t]\mathsf{eyes\text{-}sore}(a)) \end{aligned} \tag{5.2}$$

Reassignment (R) plays an important role in the solution to the frame problem [76] in TAL. Reassignment expresses change, and unless a feature is involved in reassignment, it is assumed not to change.

[1] Previously, the notation $[s,t]\mathsf{fire}(x) := T$ has been used for reassignment [90]. However, in order to be coherent with the notation for the additional operations on features that are introduced in this chapter, $R((s,t]\mathsf{fire}(x))$ has been preferred.

[2] Previously we have used integer time but TAL-C uses natural numbers.

The reassignment operator is defined as follows:[3]

$$R((\tau,\tau')\alpha) \stackrel{\text{def}}{=} (X((\tau,\tau')\alpha) \wedge [\tau']\alpha)$$
$$R([\tau]\alpha) \stackrel{\text{def}}{=} (X([\tau]\alpha) \wedge [\tau]\alpha)$$

X is an operator that represents "occlusion" of the features in α, whereas the right-most parts of the definitions denote that α holds at the end of the interval. As usual, occlusion represents an exception from the general principle of persistence, so features that are occluded are allowed to change from one time-point to the next. By minimizing occlusion, the time-points where a specific feature can change value are restricted to those time-points where the feature in question explicitly appears in a reassignment. The fact that features normally do not change is encoded in the nochange axiom below. It states that unless the feature f is occluded at time-point $t+1$, it must have the same value at $t+1$ as at time-point t.

$$\forall t, f, v[\neg X([t+1]f) \rightarrow ([t]f \hat{=} v \equiv [t+1]f \hat{=} v)] \tag{5.3}$$

To illustrate how one can reason in TAL, consider the scenario description in (5.1). From 0 to 2, there is no reassignment and therefore no occlusion, so one can with the aid of (5.3) infer

$$[0,2]\mathsf{dry}(\mathsf{wood1}) \wedge \neg\mathsf{fire}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1}).$$

From lines acs_1 and occ_1, it follows that

$$[2]\mathsf{dry}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1}) \rightarrow R((2,5]\mathsf{fire}(\mathsf{wood1}))$$

holds, which yields $[5]\mathsf{fire}(\mathsf{wood1})$. As the two other features are not occluded, due to (5.3) one has that

$$[3,5]\mathsf{dry}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1})$$

holds, and then that

$$[6]\mathsf{dry}(\mathsf{wood1}) \wedge \mathsf{fire}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1})$$

holds. From acs_2 and occ_2, it follows that

$$R((6,7]\neg\mathsf{dry}(\mathsf{wood1})) \wedge R((6,7]\neg\mathsf{fire}(\mathsf{wood1}))$$

[3] These definitions are actually encoded in the translation process from $\mathcal{L}$(ND) to $\mathcal{L}$(FL) in Appendix C. The $\mathcal{L}$(ND) versions in this section are preliminary.

holds, which yields

$$[7]\neg\mathsf{dry}(\mathsf{wood1}) \wedge \neg\mathsf{fire}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1}).$$

Finally, as nothing happens after 7,

$$[t]\neg\mathsf{dry}(\mathsf{wood1}) \wedge \neg\mathsf{fire}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1})$$

holds for all $t \geq 7$.

5.2.2 The Language $\mathcal{L}$(ND)

This section defines the surface language $\mathcal{L}$(ND) for sequential scenarios. Extensions for concurrency are introduced in Section 5.5, and the translation to the first-order language $\mathcal{L}$(FL) is presented in Appendix C. We use the overline as abbreviation of a sequence, when the contents of the sequence is obvious. For example, $f(\overline{x},\overline{y})$ means $f(x_1,...,x_n,y_1,...,y_m)$.

Definition 5.2.1 (vocabulary)
A TAL vocabulary $\nu = \langle C, F, A, T, V, R, S, \sigma\rangle$ is a tuple where C is a set of constant symbols, F is a set of feature symbols, A is a set of action symbols, T is a set of temporal function symbols, V is a set of value function symbols, R is a set of relation symbols, S is a set of basic sorts and σ is a function that maps each member of these symbol sets to a sort declaration of the form $\mathcal{S}_1 \times \ldots \times \mathcal{S}_n$ or $\mathcal{S}_1 \times \ldots \times \mathcal{S}_n \rightarrow \mathcal{S}_{n+1}$ where $\mathcal{S}_i \in S$. □

Definition 5.2.2 (basic sorts)
There are a number of sorts for values $\mathcal{V}_i$, including the boolean sort $\mathcal{B}$ with the constants $\{\mathsf{T},\mathsf{F}\}$. There are a number of sorts for features $\mathcal{F}_i$, each one associated with a value domain $dom(\mathcal{F}_i) = \mathcal{V}_j$ for some j, a sort for actions $\mathcal{A}$, and a temporal sort $\mathcal{T}$. □

The sort $\mathcal{T}$ is assumed to be interpreted, but can be axiomatized in first-order logic as a subset of Presburger arithmetic [51] (natural numbers with addition).

Definition 5.2.3 (terms)
A *value term* ω is a variable v or a constant v of sort $\mathcal{V}_i$ for some i, or an expression $\mathsf{g}(\omega_1,\ldots,\omega_n)$ where $\mathsf{g} : \mathcal{V}_{k_1} \times \ldots \times \mathcal{V}_{k_n} \rightarrow \mathcal{V}_i$ is a value function symbol and each ω_j is of sort $\mathcal{V}_{k_j}$. A *temporal term* τ is a variable t or a constant $0,1,2,3,\ldots$ or $\mathsf{s}_1,\mathsf{t}_1,\ldots$, or an expression on the form $\tau_1+\tau_2$, all of sort $\mathcal{T}$. A fluent term ϕ is a feature variable f or an expression $\mathsf{f}(\omega_1,\ldots,\omega_n)$ where $\mathsf{f} : \mathcal{V}_{k_1} \times \ldots \times \mathcal{V}_{k_n} \rightarrow \mathcal{F}_i$ is a feature symbol and each ω_j is of sort $\mathcal{V}_{k_j}$. □

Definition 5.2.4 (temporal and value formulas)
If τ and τ' are temporal terms, then $\tau = \tau'$, $\tau < \tau'$ and $\tau \leq \tau'$ are *temporal formulas*. A *value formula* is of the form $\omega = \omega'$ where ω and ω' are value terms, or $\mathsf{r}(\omega_1, \ldots, \omega_n)$ where $\mathsf{r} : \mathcal{V}_{k_1} \times \ldots \times \mathcal{V}_{k_n}$ is a relation symbol and each ω_j is of sort $\mathcal{V}_{k_j}$. □

Definition 5.2.5 (fluent formula)
An *elementary fluent formula* has the form $f \hat{=} \omega$ where f is a fluent term of sort $\mathcal{F}_i$ and ω is a value term of sort $dom(\mathcal{F}_i)$. A *fluent formula* is an elementary fluent formula or a combination of fluent formulas formed with the standard logical connectives and quantifiers. □

The elementary fluent formula $f \hat{=} \mathsf{T}$ ($f \hat{=} \mathsf{F}$) can be abbreviated f ($\neg f$).

Definition 5.2.6 (timed formulas)
Let τ, τ' be temporal terms and α a fluent formula. Then $[\tau, \tau']\alpha$, $(\tau, \tau']\alpha$ and $[\tau]\alpha$ are *fixed fluent formulas*, $C_T([\tau]\alpha)$ is a *becomes formula*, $R((\tau, \tau']\alpha)$, $R([\tau, \tau']\alpha)$and $R([\tau]\alpha)$ are *reassignment formulas*, and $X((\tau, \tau']\alpha)$, $X([\tau, \tau']\alpha)$ and $X([\tau]\alpha)$ are *occlude formulas*. Fixed fluent formulas, becomes formulas, reassignment formulas and occlude formulas are called timed formulas. □

Definition 5.2.7 (static formula)
A logical combination (including quantifiers) of temporal and value formulas, fixed fluent formulas and/or becomes formulas is called a *static formula*. □

Definition 5.2.8 (change formula)
A *change formula* is a formula that has (or is rewritable to) the form $\mathcal{Q}\overline{v}(\alpha_1 \vee \ldots \vee \alpha_n)$ where $\mathcal{Q}\overline{v}$ is a sequence of quantifiers with variables, and each α_i is a conjunction of static, occlusion and reassignment formulas. The change formula is called *balanced* iff the following two conditions hold. (a) Whenever a feature $f(\overline{\omega})$ appears inside a reassignment or occlusion formula in one of the α_i disjuncts, it must also appear in all other α_i's inside a reassignment or occlusion formula with exactly the same temporal argument. (b) Any existentially quantified variable v in the formula, whenever appearing inside a reassignment or occlusion formula, only does so in the position $f \hat{=} v$. □

Definition 5.2.9 (application formula)
An *application formula* is any of the following: (a) a balanced change formula; (b) $\Lambda \rightarrow \Delta$, where Λ is a static formula and Δ is a balanced change formula; or (c) a combination of elements of types (a) and (b) formed with $\wedge$ and $\forall$. □

Definition 5.2.10 (occurrence formula)
An *occurrence formula* has the form $[\tau, \tau']\Phi(\overline{\omega})$, where τ and τ' are temporal terms, Φ is an action name of sort $\mathcal{V}_1 \times \ldots \times \mathcal{V}_n \rightarrow \mathcal{A}$ and the value terms in $\overline{\omega}$ are of matching sorts. □

Definition 5.2.11 (scenario description components)
An action law (labeled acs) has the form $\forall t, t', \overline{x}, \overline{y}[\,[t, t']\Phi(\overline{x}) \rightarrow \Psi(\overline{x}, \overline{y})]$ where $[t, t']\Phi(\overline{x})$ is an occurrence formula and $\Psi(\overline{x}, \overline{y})$ is an application formula. A dependency law (labeled dep) has the form $\forall t, \overline{x}[\Psi(\overline{x})]$ where $\Psi(\overline{x})$ is an application formula. An observation (labeled obs) is a static formula. An occurrence (labeled occ) is an occurrence formula $[\tau, \tau']\Phi(\overline{\omega})$ where τ, τ' and $\overline{\omega}$ all are variable-free terms. □

5.3 Variations on the Concurrency Theme

In this section, we release the sequentiality assumption from the previous section and identify a number of issues that a language for scenarios with concurrency should be able to handle. We observe that TAL is sufficient for handling some of these issues, namely action duration and concurrent execution of independent actions. When it comes to interacting actions, we observe that TAL is not sufficient in a number of cases. This observation forms the basis for the discussion on how to extend TAL to handle concurrency in the subsequent sections. Notice that although we address mainly actions, the discussion also applies to dependencies.

In many domains, it is a fact that actions take time. As long as actions occur sequentially, the only way actions can interact is when the effects of one action affect the context in which a later action is executed. Therefore, it can make sense to abstract away action durations in the case of sequentiality, as is done in for instance basic situation calculus [85]. However, in the case of concurrency, the durations of actions are important for a number of reasons. First, the way the durations of two or more actions overlap can determine how they interact. Second, an action can overlap with two or more other actions without these latter actions overlapping. Third, what happens in the duration of an action can be important for how it interacts with other concurrent actions. TAL has explicit time and actions in TAL have duration.

Concurrent actions can be independent, and can involve disjoint sets of features. In this case, the combined effect of the concurrent actions is simply the union of the individual effects, as in the example below.

Scenario Description 5.4

acs_1 $[s,t]\mathsf{LightFire}(a,x) \rightarrow ([s]\mathsf{dry}(x) \wedge \mathsf{wood}(x) \rightarrow R((s,t]\mathsf{fire}(x)))$
acs_2 $[s,t]\mathsf{PourWater}(a,x) \rightarrow R((s,t]\neg\mathsf{dry}(x)) \wedge R((s,t]\neg\mathsf{fire}(x))$
obs_1 $[0]\mathsf{dry}(\mathsf{wood1}) \wedge \neg\mathsf{fire}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1})$
obs_2 $[0]\mathsf{dry}(\mathsf{wood2}) \wedge \neg\mathsf{fire}(\mathsf{wood2}) \wedge \mathsf{wood}(\mathsf{wood2})$
occ_1 $[2,7]\mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})$
occ_2 $[2,7]\mathsf{LightFire}(\mathsf{bob},\mathsf{wood2})$

Here TAL yields the conclusion $[7](\mathsf{fire}(\mathsf{wood1}) \wedge \mathsf{fire}(\mathsf{wood2}))$ as intended. Concurrency of independent actions does not pose a problem for TAL [102], nor should it for most other formalisms that do not rely on some kind of explicit frame axioms. The difficult problems arise when concurrent actions are not independent.

We address three different problems related to concurrent execution of interdependent actions. The first problem is due to the fact that the conditions under which an action is executed are not always stable, but may be altered by the effects of other concurrent actions. Consider a slight modification of (5.4), where bob pours water on the fire wood while bill is lighting the fire. The intuitive conclusion is that the wood should not be on fire at 7. We formalize this scenario in TAL by modifying some lines in the scenario description (5.4) above.[4]

Scenario Description 5.5

occ_1 $[2,6]\mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})$
occ_2 $[3,5]\mathsf{PourWater}(\mathsf{bob},\mathsf{wood1})$

The modified scenario description allows us to infer that the wood is actually on fire: $[7]\mathsf{fire}(\mathsf{wood1})$. The reason is that the effect of the $\mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})$ action is determined only by the state at time-point 2 whereas the wood does not become wet until time-point 5. Thus, only referring to the starting state in preconditions of action laws is apparently not sufficient. One needs to take into account that the conditions under which the action is executed may be altered by the direct and indirect effects of other actions while the action is going on.

The effects of one action on the conditions of another action need not always be harmful. Often, the execution of one action can enable the successful execution of another simultaneous action. For instance, turning the latch of a door might enable opening the door. Sometimes, the enabling is mutual, and two (or more) actions have synergistic effects.

[4] We only present the modified or added lines in the subsequent scenario descriptions.

A slight modification of the scenario description above illustrates the second problem, which is due to the way effects of actions are represented with reassignment. Assume that the two last lines in (5.4) are replaced with the following:

Scenario Description 5.6
occ_1 $[3,7]\mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})$
occ_2 $[3,7]\mathsf{PourWater}(\mathsf{bob},\mathsf{wood1})$

That is to say, the lighting and the pouring actions have the same duration. Now, from acs_1 and occ_1 one can infer the effect $[7]\mathsf{fire}(\mathsf{wood1})$ and from acs_2 and occ_2 one can infer the effect $[7]\neg\mathsf{fire}(\mathsf{wood1})$. Notice that these two effects are both asserted to be direct and indefeasible. Thus, the scenario becomes inconsistent. The conclusion one would like to obtain is again that the wood is not on fire.

A variation of effect interaction is when several actions affect the same feature in a cumulative way, that is, when the total effect of the actions is an aggregate of the individual effects. Reassignment represents changes to absolute values (although these values can be calculated relative to other values), and obviously, aggregation of absolute values is not very meaningful. For instance, consider the following scenario description with a box of coins.

Scenario Description 5.7 (Coin scenario with cumulative effects)
acs_1 $[s,t]\mathsf{TakeCoin}(a,b) \rightarrow [s]\mathsf{coins}(b) \mathrel{\hat{=}} (n+1) \rightarrow R((s,t]\mathsf{coins}(b) \mathrel{\hat{=}} n)$
obs_1 $[0]\mathsf{coins}(\mathsf{box1}) \mathrel{\hat{=}} 2$
occ_1 $[2,3]\mathsf{TakeCoin}(\mathsf{bill},\mathsf{box1})$
occ_2 $[2,3]\mathsf{TakeCoin}(\mathsf{bob},\mathsf{box1})$

Both occ_1 and occ_2 produce the effect $R((2,3]\mathsf{coins}(\mathsf{box1})\hat{=}1)$. Notice that this effect states that $\mathsf{coins}(\mathsf{box1})$ has the absolute value 1, and not that $\mathsf{coins}(\mathsf{box1})$ is decremented by 1. Therefore, TAL yields $[3]\mathsf{coins}(\mathsf{box1})\hat{=}1$, while the intuitive conclusion is that there are 0 coins in the box at time-point 3.

The third problem is that the conditions for two concurrent actions might interfere. In particular, actions might compete for such things as space, objects and energy. For instance, if lighting a fire requires the use of two hands, then a two-handed agent is not able to light two fires concurrently. One way of addressing this type of conflict is in terms of limited resources.

5.4 From Action Laws to Laws of Interaction

Based on the observations in the previous section, we argue that the way action laws are formulated in TAL (and many other formalisms) is not appropriate in the case of concurrency. In this section, two potential solutions are discussed. The first solution is to deal with interactions on the level of actions by allowing more expressive action laws. The second solution is to deal with interactions on the level of effects and features, by expressing effects in a less direct and absolute manner than direct reassignment of a feature. One can also imagine numerous combined approaches, for instance where conflicts can be detected on the level of features and then resolved on the level of actions, but that will not be discussed here.

5.4.1 Interaction on the Level of Actions

As the description of the effects of actions is usually centered around actions in most formalisms, it might seem like an obvious approach to also deal with concurrent interactions on the level of actions. This approach can be realized with action laws that refer to combinations of action occurrences, as in the work of Baral and Gelfond [9], Li and Pereira [59], Bornscheuer and Thielscher [13], and Reiter [86]. The following action laws, encoded in a hypothetical extended version of TAL, illustrate how one can overcome the problems in (5.5) and (5.6). Observe how acs_1 cancels the effect of LightFire in the presence of PourWater.

Scenario Description 5.8

acs_1 $[s,t]\mathsf{LightFire}(a,x) \wedge$
$\neg\exists a', s', t'[(s \leq s' < t \vee s < t' \leq t) \wedge$
$[s',t']\mathsf{PourWater}(a',x)] \rightarrow ([s]\mathsf{dry}(x) \wedge \mathsf{wood}(x) \rightarrow R((s,t]\neg\mathsf{fire}(x)))$

acs_2 $[s,t]\mathsf{PourWater}(a,x) \rightarrow R((s,t]\neg\mathsf{dry}(x)) \wedge R((s,t]\neg\mathsf{fire}(x))$

Unfortunately, this solution has a number of weaknesses. It is no longer possible to describe actions in isolation, which weakens the case for modularity in action descriptions. All potentially interacting action combinations have to be identified and explicitly put into action laws. If actions have duration, the number of such combinations is not just determined by the number of actions, but also by the number of ways two or more actions can overlap in time.

Other factors also contribute to additional complexity. If an action has several effects, then one might not want an interference regarding just one feature to neutralize all the effects of the action. Furthermore, if the action

interfered with starts before the interfering action, then the parts of the effects of the former action that are defined to occur before the starting time of the latter action should not be prevented, as this would imply causality working backwards in time.

An additional problem is that interactions are not confined to occur exclusively between the direct effects of actions, but might also involve indirect effects. Taking into consideration all potential interactions between combinations of actions and dependency laws would simply not be feasible for most nontrivial domains. Further, this would multiply all the previously mentioned complications.

5.4.2 Interaction on the Level of Features

As an alternative to an action-centered approach, we propose an approach based on the assumption that interactions resulting from concurrency are best modeled on the level of features, and not on the level of actions. The central ideas are as follows.

1. Actions provide an interface between the agent and the environment.

2. An action law does not explicitly encode the immediate effects that the action has on the state of the world. Instead, action laws encode what *influences* the action brings upon the environment.

   For instance, instead of stating $R((s,t]\mathsf{fire}(x))$ as an effect of the action $[s,t]\mathsf{LightFire}(a,x)$, one states $I((s,t]\mathsf{fire}^*(x,\mathsf{T}))$ where $\mathsf{fire}^*(x,\mathsf{T})$ represents an influence to make the feature $\mathsf{fire}(x)$ true (the I operator is similar to R, but denotes that the expression inside is true throughout the interval). An influence represents an inclination for a feature to take on a certain value or to change in a certain direction. Thus, it is more correct to consider action occurrence statements as representing action attempts that might fail to have their expected effects due to external interference rather than representing successfully executed actions.

3. Similarly, dependencies are modified to result in influences rather than actual changes.

4. The actual effects that these influences (and indirectly the actions than caused them) have on the environment are then specified in a special type of dependency laws called influence laws. For instance, $[t]\mathsf{fire}^*(x,\mathsf{T}) \rightarrow R([t]\mathsf{fire}(x))$. Generally, each feature is associated with

a number of different influences. The behavior of a feature can be specified by a number of influence laws that describe how this feature is affected by different individual influences and combinations of influences. Thus, descriptions of features and how they change due to influences play a central role in TAL-C.

This approach implies that the emphasis of the world description has shifted from actions to features, and from action laws to influence laws. Further, it can be realized with a minimum of modifications to the TAL language, and in particular the first-order nature of TAL can be retained (see Appendix C). Influence laws have the same form as dependency laws, which are already an integral part of the language, and influences can actually be represented as features. The difference between influences and other features is purely conceptual; no new syntactic or semantic constructions particular to influences are required (although some new constructs applicable to features in general will be introduced). The term "actual features" will be reserved for features that are not influences.

Notice that the use of influences as intermediaries of change serves two purposes. First, it makes it possible to avoid logical contradiction when two or more actions and dependencies affect the same feature. For instance, the combination $[5]\mathsf{fire}^*(\mathsf{wood1},\mathsf{T})$ and $[5]\mathsf{fire}^*(\mathsf{wood1},\mathsf{F})$ is logically consistent, but the combination $[5]\mathsf{fire}(\mathsf{wood1})$ and $[5]\neg\mathsf{fire}(\mathsf{wood1})$ is not. But there is more to concurrent interactions than preserving consistency. The actual outcomes of interactions need to be represented somehow, and a rich phenomenon like concurrent interactions requires a flexible representation that goes beyond the most stereotypical cases. This is supported in TAL-C by the fact that influences are first-class objects, which can be referred to in influence laws.

The use of influences in TAL-C has some resemblance to the way physical systems are often modeled. For instance, in mechanics the position/speed of a physical body is influenced by forces, and in hydraulics the flow between tanks are influenced by pressures. Notice that in physics, influences often behave cumulatively. For instance, several forces can influence an object in a mechanical system at the same time, and these forces can be aggregated using vector addition. The term "influences" is explicitly used in qualitative reasoning about physical systems by (among others) Forbus [32]. For instance, in Forbus's qualitative process theory, the expressions I+(amount-of(dest),A[flow-rate]) and I−(amount-of(source),A[flow-rate]) denote that the flow rate between two interconnected tanks influences the amount of liquid in the tanks to increase respectively decrease. If a tank t has several pipes con-

nected, then amount-of(t) will be subject to several influences. The flow rate is in turn proportional to the difference of pressure in the two tanks (flow-rate $\propto_{Q+}$ A[pressure(src)]−A[pressure(dest)]). Yet, the use of influences in TAL-C is not identical to the use in physical modeling. Although influences surely can be used in TAL-C in ways compatible with quantitative and qualitative physical modeling, they need not always faithfully represent actual physical entities or behave cumulatively.

5.5 Extending TAL to TAL-C

This section presents the differences between TAL and TAL-C. These include a new class of features, a new effect operator I and two new classes of scenario description statements.

5.5.1 Persistent and Durational Features

In reasoning about action and change, features are typically considered to be persistent. That means that they only change under special conditions, such as during the execution of an action. In order to facilitate the use of influences, TAL-C is in addition equipped with a second type of features called durational features. These normally have a default value, and they can only have a non-default value under certain circumstances, such as during the execution of an action.[5] The predicates $Per(f)$ and $Dur(f, v)$ represent that a feature is persistent respectively durational with default value v. These predicates can be augmented with a temporal argument to support features with variable behavior, such as variable default value. The default behavior of persistent features is defined as follows:[6]

$$\forall t, f, v[Per(f) \rightarrow (\neg X([t+1]f) \rightarrow ([t]f \hat{=} v \equiv [t+1]f \hat{=} v))] \tag{5.9}$$

The default behavior of durational features is defined as follows:

$$\forall t, f, v\,[Dur(f, v) \rightarrow (\neg X([t]f) \rightarrow [t]f \hat{=} v)]. \tag{5.10}$$

Notice that the distinction between persistent and durational features is in principle orthogonal to the distinction between actual features and influences,

[5]The representation of features that are only momentarily true has previously been addressed by for instance Lifschitz and Rabinov [66] and Thielscher [97].

[6]Recall that the occlusion operator X denotes that a feature is not subject to its default assumptions at a given time-point. Furthermore, the following definitions are preliminary; the final versions are presented in Appendix C.

although in practice actual features are mostly persistent and influences are mostly durational. Naturally, one need not be confined to the two types of features presented here, but they suffice for the purposes of this thesis. We should also mention that the distinction between persistent and durational features has proven useful for more than concurrency. In particular, it has proven fruitful for addressing the qualification problem [52].

5.5.2 Syntactical Additions

In addition to the reassignment operator R, we provide a new operator I which is typically (but not necessarily always) used for durational features. It is used to state that something holds over an interval.

$$I((\tau,\tau']\alpha) \stackrel{\text{def}}{=} X((\tau,\tau']\alpha) \wedge \forall t(\tau < t \leq \tau' \rightarrow [t]\alpha) \tag{5.11}$$

This specifies that the features in α are exempt from their default behaviors and that the formula α is true at all time-points from $\tau + 1$ to τ'.

The definitions of a change formula and a balanced change formula are extended to include durational formulas of the forms $I((\tau,\tau']\alpha)$, $I([\tau,\tau']\alpha)$ and $I([\tau']\alpha)$, and with the same restrictions as for R and X formulas. Two new kinds of scenario statements are introduced: Domain formulas (dom) that can contain *Per* and *Dur* elements and value formulas, and influence laws (inf), which have the same syntax as dependency laws.

5.5.3 An Example

In the following scenario description, there is a durational feature fire* representing influences on the actual feature fire. Henceforth, we will follow the convention of representing the influences on an actual feature $\mathsf{f}(\overline{\omega})$ with $\mathsf{f}^*(\overline{\omega}, v)$, where v is a value in the domain of f.

Scenario Description 5.12
dom_1 $Per(\mathsf{fire}(x)) \wedge Dur(\mathsf{fire}^*(x,v),\mathsf{F})$
acs_1 $[s,t]\mathsf{LightFire}(a,x) \rightarrow I((s,t]\mathsf{fire}^*(x,\mathsf{T}))$
inf_1 $[s,s+3]\mathsf{fire}^*(x,\mathsf{T}) \wedge \neg\mathsf{fire}^*(x,\mathsf{F}) \rightarrow R([s+3]\mathsf{fire}(x))$
inf_2 $[s]\mathsf{fire}^*(x,\mathsf{F}) \rightarrow R([s]\neg\mathsf{fire}(x))$
occ_1 $[2,6]\mathsf{LightFire}(\mathsf{Bob},\mathsf{wood1})$

Notice how acs_1 does not immediately cause fire to be true. Instead, it produces an influence to make fire true, using the I operator. How influences on fire then affect fire is described in inf_1 and inf_2. It takes 4 consecutive time-points of positive influence to make fire true, while it takes just 1 time-point of

negative influence to make it false. The influence to make fire false always has precedence over the influence to make fire true. As a matter of fact, inf_1 and inf_2 are general enough to handle any conflict that can occur between a group of actions/dependencies that try to make fire both true and false at the same time. Thus, one can in principle add arbitrary action laws and dependency laws influencing fire without any worries that they lead to inconsistency. Of course, it might still be desirable to refine or modify the way conflicts between influences are treated as a domain is elaborated and more actions and dependencies are introduced. In the examples to follow, we will continue to use influences in a manner that permits easy extension of scenarios, although this practice leads to somewhat larger scenario descriptions. This issue is further elaborated in Section 5.7.

5.6 Variations on the Concurrency Theme Revisited

In Section 5.3, a number of concurrent interactions that our original TAL formalism could not handle were identified. In this section, we show how these interactions can be represented in TAL-C. We should emphasize that although we present specific examples, the techniques employed in this section are applicable to frequently reoccurring classes of interactions.

5.6.1 Interactions from Effects to Conditions

Scenario description (5.5) was an example of the effects of one action interfering with the execution of another concurrent action. While Bill was lighting a fire, Bob poured water on the wood. This type of interference can be handled by including a suitable condition in the influence law that makes the fire feature true.

The following two laws state that the fact that the wood is not dry produces an influence $\mathsf{fire}^*(x, \mathsf{F})$ to extinguish the fire (if there is one), and that in order to affect the feature $\mathsf{fire}(x)$ the influence $\mathsf{fire}^*(x, \mathsf{T})$ for starting the fire has to be applied without interference for an extended period of time. The non-interference condition in this case is that $\mathsf{fire}^*(x, \mathsf{F})$ stays false.

$$\begin{array}{ll} \mathsf{dep}_1 & [s]\neg\mathsf{dry}(x) \rightarrow I([s]\mathsf{fire}^*(x, \mathsf{F})) \\ \mathsf{inf}_1 & [s, s+3]\mathsf{fire}^*(x, \mathsf{T}) \wedge \neg\mathsf{fire}^*(x, \mathsf{F}) \wedge \mathsf{wood}(x) \rightarrow R([s+3]\mathsf{fire}(x)) \end{array} \tag{5.13}$$

Below is the complete modified version of scenario (5.5). The action laws acs_1 and acs_2 and dependency law dep_1 produce influences, and the effects that these influences have, alone and in combination, are specified in inf_1, inf_2, inf_3 and inf_4.

Scenario Description 5.14

dom_1 $Per(\mathsf{fire}(x)) \wedge Dur(\mathsf{fire}^*(x,v),\mathsf{F})$
dom_2 $Per(\mathsf{dry}(x)) \wedge Dur(\mathsf{dry}^*(x,v),\mathsf{F})$
dom_3 $Per(\mathsf{wood}(x))$
acs_1 $[s,t]\mathsf{LightFire}(a,x) \rightarrow I((s,t]\mathsf{fire}^*(x,\mathsf{T}))$
acs_2 $[s,t]\mathsf{PourWater}(a,x) \rightarrow I((s,t]\mathsf{dry}^*(x,\mathsf{F}))$
dep_1 $[s]\neg\mathsf{dry}(x) \rightarrow I([s]\mathsf{fire}^*(x,\mathsf{F}))$
inf_1 $[s,s+3]\mathsf{fire}^*(x,\mathsf{T}) \wedge \neg\mathsf{fire}^*(x,\mathsf{F}) \wedge \mathsf{wood}(x) \rightarrow R([s+3]\mathsf{fire}(x))$
inf_2 $[s]\mathsf{fire}^*(x,\mathsf{F}) \rightarrow R([s]\neg\mathsf{fire}(x))$
inf_3 $[s,s+3]\mathsf{dry}^*(x,\mathsf{T}) \wedge \neg\mathsf{dry}^*(x,\mathsf{F}) \rightarrow R([s+3]\mathsf{dry}(x))$
inf_4 $[s]\mathsf{dry}^*(x,\mathsf{F}) \rightarrow R([s]\neg\mathsf{dry}(x))$
obs_1 $[0]\neg\mathsf{fire}(\mathsf{wood1}) \wedge \mathsf{dry}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1})$
occ_1 $[2,6]\mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})$
occ_2 $[3,5]\mathsf{PourWater}(\mathsf{bob},\mathsf{wood1})$

The fact that the wood is not on fire at 7 can be inferred as follows (we provide a first-order proof in Appendix C). Due to occ_2 and acs_2, the condition $(3,5]\mathsf{dry}^*(\mathsf{wood1},\mathsf{F})$ holds. This condition and inf_4 yield $[4,5]\neg\mathsf{dry}(\mathsf{wood1})$, and as dry is persistent, $[6]\neg\mathsf{dry}(\mathsf{wood1})$ and $[7]\neg\mathsf{dry}(\mathsf{wood1})$. The rule dep_1 then yields $[7]\mathsf{fire}^*(\mathsf{wood1},\mathsf{F})$. Finally, inf_2 gives $[7]\neg\mathsf{fire}(\mathsf{wood1})$. Notice that although $[3,6]\mathsf{fire}^*(\mathsf{wood1},\mathsf{T})$ holds, the condition $[s,s+3](\mathsf{fire}^*(\mathsf{wood1},\mathsf{T}) \wedge \neg\mathsf{fire}^*(\mathsf{wood1},\mathsf{F}) \wedge \mathsf{wood}(\mathsf{wood1}))$ does not hold for any $s \leq 3$, and this is the only condition (in inf_1) under which $\mathsf{fire}(\mathsf{wood1})$ can become true.

The case when an effect of one action enables the effect of another action can also be handled with conditional influence laws. For instance, the following influence law states that opening a door requires initially keeping the latch open (the example is originally due to Allen [4]):

inf_1 $[t]|\mathsf{latch\text{-}open} \wedge [t,t+5]\mathsf{open}^*(\mathsf{T}) \rightarrow R([t+5]\mathsf{open})$ (5.15)

A variation of enablement is when the concurrent execution of two or more actions may mutually enable a common effect that none of them could have in isolation. This phenomenon is referred to as synergistic effects. It can also be the case that the concurrent execution of several actions may prevent effects that each of the actions would have in isolation. In TAL-C, this can be achieved with the use of dependency laws. One example which contains both synergistic enablement and prevention is the scenario with a soup bowl standing on a table (the version presented here is an elaboration of the original scenario, which is due to Gelfond, Lifschitz and Rabinov [36]). The table has four sides: l for left, r for right, f for front and b for back. The variables a, x and r represent the agent, the table and a side of the table, respectively. The table can be lifted at any side (acs_1). The actual feature

lift-s(x, r) represents that the table x is lifted on side r. If the table is lifted at two opposite sides, then it is lifted from the ground (dep_1), but if is not lifted at opposite sides, then it is tilted (dep_2). If there is a soup bowl on the table and the table is tilted, then the soup is spilled (dep_3). The relation opp, defined in acc, specifies when two sides are opposite.

Scenario Description 5.16

dom_1 $Dur(\text{lift-s}(x, r), \mathsf{F}) \wedge Dur(\text{lift-s}^*(x, r, v), \mathsf{F})$
dom_2 $Dur(\text{tilted}(x), \mathsf{F}) \wedge Dur(\text{tilted}^*(x, v), \mathsf{F})$
dom_3 $Dur(\text{lifted}(x), \mathsf{F}) \wedge Dur(\text{lifted}^*(x, v), \mathsf{F})$
dom_4 $Per(\text{spilled}(x)) \wedge Dur(\text{spilled}^*(x, v))$
dom_5 $Per(\text{soup}(x)) \wedge Per(\text{table}(x)) \wedge Per(\text{on}(x, y))$
acc $\text{opp}(r, r') \equiv [\langle r, r' \rangle = \langle \mathsf{l}, \mathsf{r} \rangle \vee \langle r, r' \rangle = \langle \mathsf{r}, \mathsf{l} \rangle \vee \langle r, r' \rangle = \langle \mathsf{f}, \mathsf{b} \rangle \vee \langle r, r' \rangle = \langle \mathsf{b}, \mathsf{f} \rangle]$
acs_1 $[s, t]\text{Lift}(a, x, r) \rightarrow I((s, t]\text{lift-s}^*(x, r, \mathsf{T}))$
dep_1 $[t](\text{table}(x) \wedge \exists r_1, r_2[\text{lift-s}(x, r_1) \wedge \text{lift-s}(x, r_2) \wedge \text{opp}(r_1, r_2)]) \rightarrow$
$I([t]\text{lifted}^*(x, \mathsf{T}))$
dep_2 $[t](\text{table}(x) \wedge \text{lift-s}(x, r_1) \wedge \neg\exists r_2, r_3[\text{lift-s}(x, r_2) \wedge \text{lift-s}(x, r_3) \wedge$
$\text{opp}(r_2, r_3)]) \rightarrow I([t]\text{tilted}^*(x, \mathsf{T}))$
dep_3 $[t]\text{tilted}(y) \wedge \text{on}(x, y) \wedge \text{soup}(x) \rightarrow R([t + 1]\text{spilled}^*(x, \mathsf{T}))$
inf_1 $[t]\text{lift-s}^*(x, r, \mathsf{T}) \rightarrow I([t]\text{lift-s}(x, r))$
inf_2 $[t]\text{tilted}^*(x, \mathsf{T}) \rightarrow I([t]\text{tilted}(x))$
inf_3 $[t]\text{spilled}^*(x, \mathsf{T}) \rightarrow I([t]\text{spilled}(x))$
inf_4 $[t]\text{lifted}^*(x, \mathsf{T}) \rightarrow I([t]\text{lifted}(x))$
obs_1 $[0]\text{table}(\mathsf{t1}) \wedge \text{soup}(\mathsf{s1}) \wedge \text{on}(\mathsf{s1},\mathsf{t1})$
occ_1 $[3, 6]\text{Lift}(\mathsf{bill},\mathsf{t1},\mathsf{l})$
occ_2 $[3, 6]\text{Lift}(\mathsf{bob},\mathsf{t1},\mathsf{r})$

In this scenario, one can infer from the statements occ_1, occ_2, inf_1 and inf_2 that lift-s(t1, l) $\wedge$ lift-s(t1, r) holds from 4 to 6. Thus, the condition $\exists r_1, r_2[\text{lift-s}(x, r_1) \wedge \text{lift-s}(x, r_2) \wedge \text{opp}(r_1, r_2)]$ is satisfied in this time interval, enabling the effect lifted(t1) from dep_1 and inf_4, while preventing the effect tilted(t1) according to dep_2 and inf_2.

The table lifting scenario encodes several other potential interactions between lifting actions. If the two lifting actions are not synchronized, the table is tilted and the soup is spilled. For instance, if occ_2 is altered to $[4, 7]$Lift(bob,t1,r), then one can infer both $[4](\text{table}(\mathsf{t1}) \wedge \text{lift-s}(\mathsf{t1}, \mathsf{l})$ and $\neg\exists r_2, r_3[\text{lift-s}(\mathsf{t1}, r_2) \wedge \text{lift-s}(\mathsf{t1}, r_3) \wedge \text{opp}(r_2, r_3)])$. According to dep_2 and inf_2, this produces the effect $[4]$tilted(t1), and the soup is spilled. Also notice that if the table is lifted from three sides, (for instance, add occ_3 $[4, 7]$Lift(ben,t1,f) to the scenario description) then the condition $\exists r_1, r_2[\text{lift-s}(x, r_1) \wedge \text{lift-s}(x, r_2) \wedge$

$\mathsf{opp}(r_1, r_2)$] is still satisfied, which implies that the table is lifted and not tilted. However, if the only occurrences are occ_1 and occ_3, then that condition does not hold and the table is tilted and the soup is spilled.

Notice that it is possible to write influence laws that determine directly from the lift-s* influence whether the table is lifted or the soup is spilled. Anyhow, we have preferred to explicitly represent the causal chain from lifting to spilling and tilting, as this makes it easier to extend the scenario to include actions that for instance counter-act the lifting by pressing down a side of the table or that stabilize the bowl.

5.6.2 Interactions Between Effects

The previous subsection addressed interactions between effects and conditions of actions and dependencies. As observed in Section 5.3, another problem is when two or more actions or dependencies are affecting the same feature. Such combinations of effects can be conflicting or cumulative.

Conflicting effects

Returning to the fire lighting scenario description (5.14), it can be observed that the use of influence laws in that scenario also solves the problem of conflicting effects that was observed in (5.6) in Section 5.3. There were two influence laws in (5.14) that determined the result of conflicting influences on the fire feature:

$$\mathsf{inf}_1 \quad [s, s+3]\mathsf{fire}^*(x, \mathsf{T}) \wedge \neg\mathsf{fire}^*(x, \mathsf{F}) \wedge \mathsf{wood}(x) \rightarrow R([s+3]\mathsf{fire}(x)) \qquad (5.17)$$
$$\mathsf{inf}_2 \quad [s]\mathsf{fire}^*(x, \mathsf{F}) \rightarrow R([s]\neg\mathsf{fire}(x))$$

Now assume that the occurrences in (5.14) are modified as follows.

$$\mathsf{occ}_1 \quad [2,6]\mathsf{LightFire}(\mathsf{bill}, \mathsf{wood1}) \qquad (5.18)$$
$$\mathsf{occ}_2 \quad [4,6]\mathsf{PourWater}(\mathsf{bob}, \mathsf{wood1})$$

In this case, one can infer $(2,6]\mathsf{fire}^*(\mathsf{wood1}, \mathsf{T})$, which normally has the effect $[6]\mathsf{fire}(\mathsf{wood1})$ (inf_1). One can also infer $[6]\neg\mathsf{dry}(\mathsf{wood1})$ and $[6]\mathsf{fire}^*(\mathsf{wood1}, \mathsf{F})$, which normally results in $[6]\neg\mathsf{fire}(\mathsf{wood1})$ (inf_2). The conflict between these two influences is resolved in inf_1 and inf_2, to the advantage of the latter.

The fire lighting scenario illustrates a conflict involving just two opposite influences. However, there might also be conflicts involving arbitrarily large numbers of influences. For instance, consider the following scenario where several agents try to pick up the same object. The feature $\mathsf{pos}(x)$ represents the position of an object x, and its value domain of positions includes both locations (e.g. floor) and agents (e.g. bill, bob). An object can only have

one position, so when more than one agent is trying to take the object, then there is a conflict. This conflict is resolved in inf_1 in the scenario description.

Scenario Description 5.19

dom_1 $Dur(\mathsf{pos}^*(x,a),\mathsf{F}) \wedge Per(\mathsf{pos}(x))$
acs_1 $[s,t]\mathsf{Pickup}(a,x) \rightarrow I((s,t]\mathsf{pos}^*(x,a))$
inf_1 $[t]\mathsf{pos}(x)\hat{=}\mathsf{floor} \wedge \exists p[\,[t+1]\mathsf{pos}^*(x,p)\,] \rightarrow$
$\exists p[\,[t+1]\mathsf{pos}^*(x,p) \wedge R([t+1]\mathsf{pos}(x)\hat{=}p)\,]$
obs_1 $[0]\mathsf{pos}(\mathsf{wallet}) \hat{=} \mathsf{floor}$
occ_1 $[2,3]\mathsf{Pickup}(\mathsf{bill},\mathsf{wallet})$
occ_2 $[2,3]\mathsf{Pickup}(\mathsf{bob},\mathsf{wallet})$

The line labeled inf_1 states that "if the object x is on the floor and at least one agent is trying to take x then one of the agents who are trying to take x will actually have x". The result in this case is nondeterministic, and this is perhaps the best way to treat conflicts when one lacks detailed information of what the actual result would be. Notice that the consequent of inf_1 only changes the value of $\mathsf{pos}(x)$, and not the value of the influence $\mathsf{pos}^*(x,p)$. The $\mathsf{pos}^*(x,p)$ component of the consequent is in effect a filter on the values p that $\mathsf{pos}(x)$ can be reassigned to.

It is equally possible to state that no effect occurs in the case of conflict:

$$\mathsf{inf}_1 \quad ([t]\mathsf{pos}(x)\hat{=}\mathsf{floor} \wedge [t+1]\mathsf{pos}^*(x,p) \wedge \neg\exists p'[\,[t+1]\mathsf{pos}^*(x,p') \wedge p \neq p'\,]) \rightarrow R([t+1]\mathsf{pos}(x)\hat{=}p) \tag{5.20}$$

Finally, some values might be preferred to other values. For instance, we can enhance the picking-up scenario by giving preference to stronger agents, as follows. The relation stronger encodes a partial ordering on agents based on their relative strength.

$$\begin{aligned}\mathsf{inf}_1 \quad & [t]\mathsf{pos}(x)\hat{=}\mathsf{floor} \wedge \exists p[\,[t+1]\mathsf{pos}^*(x,p)\,] \rightarrow \\ & \exists p[\,[t+1]\mathsf{pos}^*(x,p) \wedge \neg\exists p'[\,[t+1]\mathsf{pos}^*(x,p') \wedge \mathsf{stronger}(p',p)\,] \wedge \\ & R([t+1]\mathsf{pos}(x)\hat{=}p)\,]\end{aligned} \tag{5.21}$$

Cumulative effects

Another common phenomenon besides conflicting influences is when influences signify some relative change, and therefore multiple influences can be combined in a cumulative way.

We have already mentioned that in mechanics, multiple forces on an object can be combined using vector addition, and the vectorial sum determines changes in the objects speed and position. Here, we present another example, which involves a box from which agents can take coins. In order to specify

cumulative effects, we need to introduce a minimal portion of set theory, including set membership (*in*), the empty set (*empty*) and subtraction of one element from a set (*remove*). A set theory that is sufficient for our purpose is obtained using the following two axioms. The sets contain only features of a specific feature sort, so the axioms have to be restated for different sorts. The variable σ represents sets.

$$\forall \sigma, f, f'[in(f, remove(f', \sigma)) \equiv (in(f, \sigma) \wedge f \neq f')] \tag{5.22}$$

$$\forall \sigma[empty(\sigma) \equiv \forall f[\neg in(f, \sigma)]\,] \tag{5.23}$$

Furthermore, we provide the following definition of the sum of feature values over a set of features.

$$\begin{array}{l} \forall t, \sigma, f, m, n[in(f, \sigma) \wedge sum(t, remove(\sigma, f)) = m \wedge [t]f \hat{=} n \rightarrow \\ \qquad\qquad sum(t, \sigma) = (m + n)] \\ \forall t, \sigma[empty(\sigma) \rightarrow sum(t, \sigma) = 0] \end{array} \tag{5.24}$$

Now we can introduce an influence $\mathsf{coins}^-(a, c)$ with a value domain of natural numbers and default value 0 to represent that an agent a is taking a coin from a container c. In addition, we define the special function $\mathsf{Coins}^-(t, c)$ which for a given c represents the set of all $\mathsf{coins}^-(a, c)$ with a nonzero value at time-point t.

$$in(\mathsf{coins}^-(a, c'), \mathsf{Coins}^-(t, c)) \equiv (c = c' \wedge [t]\neg\mathsf{coins}^-(a, c) \hat{=} 0) \tag{5.25}$$

This definition establishes the existence of a set which contains all the nonzero features of the relevant type. The definition of *remove* above then establishes the existence of all subsets of this set, which is sufficient for determining the sum of the values of all features in the set.

The scenario description (5.7) is modified as follows.

Scenario Description 5.26

dom_1 $Dur(\mathsf{coins}^-(a, c), 0) \wedge Per(\mathsf{coins}(c))$
acs_1 $[s, t]\mathsf{TakeCoin}(a, c) \rightarrow I([t]\mathsf{coins}^-(a, c) \hat{=} 1)$
inf_1 $[t]\mathsf{coins}(c)\hat{=}(m + n) \wedge sum(t+1, \mathsf{Coins}^-(t + 1, c)) = n) \rightarrow$
$R([t + 1]\mathsf{coins}(c)\hat{=}m$
obs_1 $[0]\mathsf{coins}(\mathsf{box1})\hat{=}2$
occ_1 $[2, 3]\mathsf{TakeCoin}(\mathsf{bill}, \mathsf{box1})$
occ_2 $[2, 3]\mathsf{TakeCoin}(\mathsf{bob}, \mathsf{box1})$

The cumulative behavior of the coins feature is encoded in the influence statement inf_1, which adds together the values of all non-zero $\mathsf{coins}^-(a, c)$ for a specific c and adds this sum to $\mathsf{coins}(c)$. From this scenario description, one can infer that $\mathsf{Coins}^-(3, \mathsf{box1}) = \{\mathsf{coins}^-(\mathsf{bill}, \mathsf{box1}), \mathsf{coins}^-(\mathsf{bob}, \mathsf{box1})\}$. As $[3]\mathsf{coins}^-(\mathsf{bill}, \mathsf{box1}) \hat{=} 1$ and $[3]\mathsf{coins}^-(\mathsf{bob}, \mathsf{box1}) \hat{=} 1$ we get the result that $sum(\mathsf{Coins}^-(3, \mathsf{box1})) = 2$ and $[3]\mathsf{coins}(\mathsf{box1}) \hat{=} 0$. Obviously, the scenario can be enhanced. For example, more than one coin can be taken by the same agent, coins can be added to the box, and so on.

5.6.3 Interacting Conditions

Finally, there is the problem that the conditions of two or more actions may interact. A special case of this is when an agent has a resource that can only be used for one action at a time. For instance, people generally have only two hands, and thus cannot simultaneously perform two actions that each requires the use of both hands, like lighting a fire. A strategy for representing resources is to introduce a feature representing what action the resource is actually used for. In the following scenario, the feature $\mathsf{uses\text{-}hands}(x)$ fulfills this function. There is a value sort of action tokens that duplicates the action sort (for example the value $\mathsf{light\text{-}fire}(a, x)$ corresponds to the action $\mathsf{LightFire}(a, x)$). The feature $\mathsf{uses\text{-}hands}(x)$ has action tokens as domain, and the letter e is used for action token variables. The action token noop stands for "no operation".

The use of a resource for an action is divided into two steps. First, the action claims the resource. The statement labeled acs_1 in the scenario description below states that the action of lighting a fire needs the "hands" resource. This need is represented by the influence $\mathsf{uses\text{-}hands}^*(a, e)$. Second, the resource is actually used and the action produces some effect. The row labeled dep_1 below states that if the hands are used for lighting a fire, then this will produce a fire influence.

Scenario Description 5.27

dom_1 $Per(\mathsf{fire}(x)) \wedge Dur(\mathsf{fire}^*(x, v), \mathsf{F})$
dom_2 $Per(\mathsf{wood}(x))$
dom_3 $Dur(\mathsf{uses\text{-}hands}(a), \mathsf{noop}) \wedge Dur(\mathsf{uses\text{-}hands}^*(a, e), \mathsf{F})$
acs_1 $[s, t]\mathsf{LightFire}(a, x) \rightarrow I((s, t]\mathsf{uses\text{-}hands}^*(a, \mathsf{light\text{-}fire}(a, x)))$
dep_1 $[t]\mathsf{uses\text{-}hands}(a) \hat{=} \mathsf{light\text{-}fire}(a, x) \rightarrow I([t]\mathsf{fire}^*(x, \mathsf{T}))$
inf_1 $\exists e[\,[t]\mathsf{uses\text{-}hands}^*(a, e)\,] \rightarrow \exists e[\,[t]\mathsf{uses\text{-}hands}^*(a, e) \wedge I([t]\mathsf{uses\text{-}hands}(a) \hat{=} e)\,]$
inf_2 $[s, s + 3]\mathsf{fire}^*(x, \mathsf{T}) \wedge \neg\mathsf{fire}^*(x, \mathsf{F}) \wedge \mathsf{wood}(x) \rightarrow R([s + 3]\mathsf{fire}(x))$
inf_3 $[s]\mathsf{fire}^*(x, \mathsf{F}) \rightarrow R([s]\neg\mathsf{fire}(x))$

obs_1 $[0]\mathsf{dry}(\mathsf{wood1}) \wedge \neg\mathsf{fire}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1})$
occ_1 $[2,6]\mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})$
occ_2 $[2,6]\mathsf{LightFire}(\mathsf{bill},\mathsf{wood2})$

The distribution of resources is encoded in inf_1, which states that if at least one action e needs the "hands" resource then some action that needs that resource will have it (compare to inf_1 in (5.19)). If two or more actions need a resource, then only one of them will have it, and the resource can randomly alter between competing actions. In this example, the value of $\mathsf{uses\text{-}hands}(a)$ alternates randomly between $\mathsf{light\text{-}fire}(\mathsf{bill},\mathsf{wood1})$ and $\mathsf{light\text{-}fire}(\mathsf{bill},\mathsf{wood2})$ from 3 to 6, with the result that both actions fail or only one of them succeeds.

More sophisticated forms of resources than the binary resource above are also possible. For instance, one can utilize the techniques presented in connection with cumulative effects to deal with quantitative and sharable resources, and resources can be renewable or consumable.

5.6.4 Special vs. General Influences

In all examples so far, each feature type[7] has its own set of influences and influence laws. While offering a high level of freedom in handling concurrent interactions, this approach also requires declaring influences for each feature type and writing down a large amount of influence laws. Of course, if one desires a flexible and non-stereotypical treatment of interactions, this is hard to avoid. But TAL-C is also capable of a more uniform and compact treatment of interactions. Instead of declaring separate influences of each actual feature type (e.g. dry^* for dry), one can group together features with the same value domain and the same behavior and let them be of the same feature sort $\mathcal{F}_i$. Next, one introduces a function $* : \mathcal{F}_i \times dom(\mathcal{F}_i) \rightarrow \mathcal{F}_j$ that for a given feature and value represents an influence on the feature to change according to the value (e.g. $*(\mathsf{dry}(\mathsf{wood1}),\mathsf{F})$). Thereby, it is possible to specify influence laws that apply to all feature types that are of the sort $\mathcal{F}_i$. The following is an example of a general influence law where $dom(\mathcal{F}_i)$ is the boolean value sort. The variable f (implicitly universally quantified) is of sort $\mathcal{F}_i$. The influence law handles conflicts by making the outcome nondeterministic.

$$\begin{array}{ll} \mathsf{dom} & Dur(*(f,v),\mathsf{F}) \\ \mathsf{inf} & ([t]*(f,\mathsf{T}) \wedge \neg*(f,\mathsf{F}) \rightarrow R([t]f)) \wedge \\ & ([t]\neg*(f,\mathsf{T}) \wedge *(f,\mathsf{F}) \rightarrow R([t]\neg f)) \wedge \\ & ([t]*(f,\mathsf{T}) \wedge *(f,\mathsf{F}) \rightarrow X([t]f)) \end{array} \tag{5.28}$$

[7] We consider all features with the same feature symbol, for example fire, to define a type. Several feature types with the same value domain might be of the same sort.

This approach yields a number of influence laws that is proportional to the number of feature sorts, instead of proportional to the number of feature types. It can be particularly useful for scenarios with a large number of feature types that exhibit relatively uniform behaviors. In addition, the fact that influence laws are more general and apply to sorts rather than to specific feature types implies a higher degree of reusability.

5.7 Working with TAL-C Scenarios

When encoding a scenario in TAL-C, a bottom-up approach involving the four following levels can be used.

1. Identify relevant features of the world and their value domains.
2. For each feature, determine its normal (non-influenced) behavior, its potential influences and how these affect the feature alone and in combination.
3. Identify actions and dependencies in the world and how these influence features.
4. Determine what holds and occurs in the world, and what individuals there are.

Often, it is not possible to work strictly sequentially from level 1 to level 4. The elaboration of a complex scenario is an iterative and incremental process, where the four levels above are intertwined and decisions made earlier need to be reconsidered. Therefore, to estimate how demanding this elaboration process would be in TAL-C, it is relevant to analyze what implications additions or modifications at different levels would have on an existing scenario description as a whole. Although there are some initial studies on the subject [75], there exist no systematic methods for performing this kind of estimate. Therefore, the following observations are based mainly on practical experience and common sense.

To provide a background for the discussion to follow, we need to make some additional assumptions about the form of a scenario description. We should emphasize that these assumptions represent good conventions that have been followed in this chapter, but they are not formally part of the TAL-C definition. An action law for an action A has the following form, where $\Lambda^{\mathsf{A}(\overline{x})}$ contains only actual features and $\Delta^{\mathsf{A}(\overline{x})}$ contains only influences:

$$\mathsf{acs} \quad [t,t']\mathsf{A}(\overline{x}) \rightarrow (\Lambda^{\mathsf{A}(\overline{x})} \rightarrow \Delta^{\mathsf{A}(\overline{x})}) \tag{5.29}$$

A dependency law has the following form, with the corresponding restrictions on $\Lambda_k(\overline{x})$ and $\Delta_k(\overline{x})$:

$$\mathsf{dep} \quad \Lambda_k(\overline{x}) \rightarrow \Delta_k(\overline{x}) \tag{5.30}$$

A domain statement for a specific feature has one of the two following forms:

$$\mathsf{dom} \quad \mathit{Per}(\mathsf{f}(\overline{x})) \tag{5.31}$$
$$\mathsf{dom} \quad \mathit{Dur}(\mathsf{f}(\overline{x}), v)$$

Finally, each feature type f has a number of influence laws of the form

$$\mathsf{inf}_i \quad \Delta_i^{\mathsf{f}}(\overline{x}) \rightarrow \Lambda_i^{\mathsf{f}}(\overline{x}) \tag{5.32}$$

where the consequent $\Lambda_i^{\mathsf{f}}(\overline{x})$ contains references to no other actual feature but $\mathsf{f}(\overline{x})$ and this feature occurs only inside reassignment, interval and occlude formulas. $\Lambda_i^{\mathsf{f}}(\overline{x})$ may also contain influences that belong to $\mathsf{f}(\overline{x})$, but then only inside static subformulas. The antecedent $\Delta_i^{\mathsf{f}}(\overline{x})$ contains only influences that belong to $\mathsf{f}(\overline{x})$ (for example $\mathsf{f}^{*(\overline{x},v)}$) and actual features. Each group of influence laws represents a module that describes the behavior of a specific feature f together with the dom statement for that feature, and any conflicts or other interactions are handled locally within that module. Finally, we assume that each influence only belongs to one actual feature.

Given the assumptions above, we can draw the following conclusions about the impact an addition or modification will have on a scenario description.

Adding a new feature (level 1 according to the enumeration above) does not in itself affect anything else. It is obviously followed by adding new influences and influence laws (level 2) and sometimes also adding or modifying actions and dependencies (level 3). These operations are discussed below.

Adding or altering the default behavior of a feature (level 2) is local to the domain statement specifying the default behavior in question. Adding a new type of influence for a feature implies altering the influence laws for the feature in question, but does not affect the influence laws for other features. If care is taken in the choice of influences, additions of influences should seldom occur, and the types of influences for a given feature should remain more or less constant. As a parallel, the physical property of an object's speed can be influenced by a large amount of imaginable actions and conditions. Yet, one single type of influence ("force") suffices to determine changes in speed. Further, altering the interactions between influences for a particular feature is local to the influence laws (that is, the module) of that feature, but does not affect the default behavior or any action or dependency laws.

Modifying or adding an action law or dependency law (level 3) does not affect any other existing action or dependency laws as any interactions are

delegated to the influence laws, nor does it affect existing influence laws. The exception is of course when the new action or dependency law requires the introduction of a new type of influence for a particular feature, in which case the influence laws of that feature have to be extended. Finally, adding action occurrences and observations (level 4) does not affect anything at the preceding levels.

In summary, additions or modifications are in general local operations which preserve modularity in TAL-C. Different features can be described in isolation, and given a set of features and associated influences, different actions and dependencies can be described in isolation. This property is mainly due to two features of the logic, namely that interactions between actions and dependencies are channeled through influences, and further that the respective sets of influences of different features are disjoint, and therefore the behaviors of features can be specified in normal behavior and influence laws that are independent of those of other features. It is encouraging to achieve this level of modularity, considering the fact that we are addressing complicated causal dependencies and concurrent interactions.

5.8 Other Work on Concurrency

Hendrix Hendrix's work [45] is an early attempt to represent continuous and simultaneous processes, using a STRIPS-like [30] language. Unlike STRIPS, Hendrix's formalism involves notions of explicit time, duration, and simultaneous and extraneous activity. A process has preconditions and continuation conditions that determine when the process can be initiated and for how long it goes on. Hendrix distinguishes between instantaneous effects at the initiation and termination moments of the process and gradual effects that occur while the process is going on. However, there are no means for determining what happens when more than one process affects the same feature ("parameter" in Hendrix's terminology). More sophisticated representations of physical processes were later developed in qualitative reasoning, where Forbus [32] has already been mentioned.

Georgeff In the work of Georgeff [37] there are world states that are linked together in histories. In a world state, features can hold and one or more events (actions) can occur. Specific features can explicitly be declared to be independent of specific events, and a persistence axiom, similar to the nochange axiom in TAL, states that if a feature p is independent of all events in a state, then it will not have changed in the next state. Georgeff then in-

troduces the concept of *correctness conditions*. If the correctness condition p of an event e is independent of another event e', then e' will not interfere with (prevent) e. Thus, Georgeff's formalism can define when two events (actions) can and cannot occur simultaneously, and what the result is when two independent events are executed simultaneously. A limitation is the lack of an explicit notion of duration, so events cannot overlap partially. Georgeff also considers processes, which are essentially related groups of events with limited interaction with events outside the process.

Lansky Structural relations between events is the central theme in work by Lansky [56]. In GEM, there is an explicit representation of event location; events can belong to elements, which are loci of forced sequential activity, and which in turn can belong to groups. In essence, groups represent boundaries of causal access. Events inside a group can only interact with external events via specific ports (causal holes). Thereby, the possible concurrent interactions between events can be restricted.

Pelavin Pelavin's work on a logic for planning with simultaneous actions with duration [82] is based on Allen's interval temporal logic [4], in which properties and actions are associated with intervals of time. Pelavin's formalism has quite a complex non-standard semantics, where the central entities are world histories. A closeness function defines how the addition of actions to a world history results in new world histories. On the syntactic level, there are modal operators on world histories: $IFTRIED(pi, P)$ denoting that the condition P would hold if the actions in pi are executed, and $INEV(i, P)$ stating that the condition P inevitably holds at time i (is independent of anything happening after i). These operators can be used for quite sophisticated descriptions of actions, including interference where one action prevents another and cumulative effects. However, what does not change due to an action has to be explicitly encoded, and there is no concept of dependency laws.

Thielscher Thielscher [97] presents a theory of dynamic systems, where state transitions can occur naturally in addition to being caused by actions. Fluents are divided into two sets. There are *persistent fluents*, which are subject to inertia and only change when directly influenced, and there are *momentary fluents* that become false if nothing affects them. A subset of the momentary fluents are the *action fluents*. Causal laws are specified in a STRIPS-style manner, with a precondition, a set of persistent fluents to

become true, a set of persistent fluents to become false and a set of momentary fluents to become true in the following state. Thielscher addresses some aspects of concurrency, but a versatile way of handling time is lacking. Durations and delays cannot be easily modeled, due to the STRIPS-style operational nature of the language. The paper also discusses one type of concurrent conflicts the formalism can handle, but no general way to handle other types of concurrent conflicts is mentioned.

Ferber and Müller Ferber and Müller [29] presents a theory for dynamic multi-agent environments with a distinction between influences and state. The world develops in two-step cycles: there is a set of operators (corresponding to actions and events) that for given influences and conditions on the state yield new influences, and a set of laws that for given conditions on the state and influences transform the state. The state component develops according to a persistence assumption, whereas influences are transient (like persistent respectively durational features in TAL-C). The theory is then augmented with agent behaviors, which are functions from influence sets (percepts) to influence sets (responses). A STRIPS-style operational formalism is used in the paper, but the authors explain that the general principles should apply to other types of formalisms as well.

Pinto and other work with the situation calculus Among the work done in situation calculus, Pinto's modeling of concurrency [83] is particularly interesting. Pinto addresses the use of resources and exploits state constraints (of a weaker kind than the dependency laws in this thesis) to deal with effect interaction, although in the context of instantaneous actions. Also other authors such as Gelfond, Lifschitz and Rabinov [36], Lin and Shoham [70] and Reiter [86] address concurrency in the context of the situation calculus. However, most of the problems these authors consider are specific to the situation calculus, including how to extend the result function to take more than one action, and how to represent action duration [36, 86] and extraneous actions [86]. The topic of concurrent interaction is only briefly addressed; for instance, Lin's and Shoham's treatment is restricted to effecting cancellation in case two actions are in conflict.

Baral and Gelfond Finally, we should also mention Baral and Gelfond's propositional language $\mathcal{A}_C$ [9] and its relatives by Li and Pereira [59] and Bornscheuer and Thielscher [13]. $\mathcal{A}_C$, an extension of Gelfond's and Lifschitz's language $\mathcal{A}$ [35], relies on action rules (e-propositions) of the form

$\{A_1, \ldots, A_n\}$ **causes** e **if** $p_1, \ldots, p_n$ to describe concurrent interactions (recall the discussion in Section 5.4 regarding some limitations of this type of approach). Rules for the same fluent with more specific action parts override less specific ones. This makes $\mathcal{A}_C$ suitable for representing synergistic and conflicting effects. On the other hand, actual cancellation of effects requires the use of explicit frame axioms, like in the soup bowl example [9] where the rule $\{$*lift_left*, *lift_right*$\}$ **causes** $\neg$*spilled* if $\neg$*spilled* overrides the rules $\{$*lift_left*$\}$ **causes** *spilled* and $\{$*lift_right*$\}$ **causes** *spilled*. None of the articles [9, 59, 13] address ramification or action duration, and only Bornscheuer's and Thielscher's version addresses nondeterminism. The three languages differ mainly in the treatment of concurrent conflicts that are not resolved by any e-proposition. According to Baral and Gelfond, the entire resulting state is undefined, according to Li and Pereira, the result of the conflicting actions is undefined, and according to Bornscheuer and Thielscher, the resulting value of the conflicting feature is nondeterministic.

5.9 Conclusions

In this chapter, we have presented TAL-C, which is a logic for describing action scenarios that involve action concurrency and causal dependencies between features. What distinguishes TAL-C from previous work is the combination of the following factors: (a) TAL-C has a standard first-order semantics and proof theory. (b) TAL-C has a notion of explicit time, which makes it possible to reason about the durations of actions and other interesting temporal properties and relations. (c) TAL-C is able to model a number of important phenomena related to concurrency. We should also mention that several of the examples in this chapter have been tested using a partial implementation of TAL and TAL-C, called VITAL (available on the WWW at `http://www.ida.liu.se/~jonkv/vital.html` as a Java program).

Technically, TAL-C is closely related to TAL with ramification [42], although the surface language $\mathcal{L}$(ND) has been modified and extended. In the base language $\mathcal{L}$(FL), the same predicates are still used, with the addition of *Per* and *Dur*. Most important, the same simple circumscription policy is still applicable, which implies that we can reason about concurrent interactions in first-order logic.

The main difference between TAL and TAL-C is conceptual in nature and based on how action laws are defined and used. We have demonstrated how traditional action laws suffer from a number of problems, in particular due to the fact that preconditions in action laws refer to the state before the

action is executed, and that the effects are absolute and indefeasible. The solution involving action laws with multiple actions was rejected due to lack of precision and scaling. Instead, we proposed an approach where actions produce influences instead of actual effects. The way these influences change the world, both alone and in interaction with other influences can then be specified in influence laws for individual features. TAL-C has been demonstrated on a number of nontrivial concurrency-related problems. The use of influence laws in TAL-C provides a flexible tool for describing what happens when a feature is subject to influence from several actions or dependencies simultaneously. Additions and modifications to a TAL-C scenario description are local operations which preserve modularity.

In the next chapter we will show how the methods developed to solve concurrency problems directly can be applied to handle many problems with delayed effects of actions and dependency rules.

Chapter 6

Delayed Effects of Actions

A fundamental property of many dynamical systems is that effects of actions can occur with some delay. In this chapter, based on the work by Karlsson, Gustafsson and Doherty [18, 49], we address the representation of delayed effects in the context of reasoning about action and change. We discuss how delayed effects can be modeled both in abstract ways and as detailed processes, and we consider a range of possible interactions between the delayed effects of an action and actions occurring at later time-points, including interference and cumulative effects.

6.1 Introduction

A fundamental property of many dynamical environments, in particular natural ones, is that changes as a response to actions or events do not occur instantaneously, but after some duration of time. This observation is obviously of great interest in reasoning about action and change, and is usually discussed in terms of delayed effects or processes. In fact, if some change occurs after the end of an action, then there must be some underlying process going on. However, a description of this process might be too complicated or not even available. By instead considering the changes caused by the process as delayed effects of the action in question, one can sometimes abstract away from the details of the process and still be able to obtain an adequate description of its manifestations. In this chapter, we show how delayed effects can be modeled in the language TAL-C described in the previous chapter.

There are three factors that make TAL-C a suitable instrument for dealing with delayed effects: the existence of a notion of time which is independent of actions; the possibilities to define causal dependencies outside of action de-

scriptions; and the support for concurrent interactions. Explicit time makes it possible to state that a delayed effect occurs after some (possibly indeterminate) amount of time, and causal dependencies and concurrent interactions are crucial in describing interactions between actions. The fact that the effects of an action are not confined temporally within the duration of that action creates ample opportunities for interactions to occur: a delayed effect of an action can prevent or be prevented by the effects of a later action, or might occur simultaneously with a later action.

Delayed effects have not received much attention in the literature. Some papers on continuous change, for example by Sandewall [90] and Shanahan [92], address the explicit representation of continuous processes that can go on after an action has been executed, such as water filling a sink after the tap has been turned on. Shanahan also presents an example with an alarm that goes off after a fixed time interval. Further, there is work by Doherty and Gustafsson [18] on representing delays with dependency laws. However, none of the logics in these papers deal with concurrent interactions. It is the modeling of how the delayed effect of one action can interact with other actions that is the most important contribution of this chapter.

An example of a delayed effect, due to Gelfond, Lifschitz and Rabinov [36], is a pedestrian light that turns green 30 seconds after one presses the button at the crosswalk. They represent the delayed effect as a postcondition of a Press action. In TAL-C, the same example (somewhat extended) is encoded in the scenario description below

Scenario Description 6.1

dom_1 $Dur(\mathsf{pressed}, \mathsf{F}) \wedge Per(\mathsf{tick}) \wedge Per(\mathsf{color})$
acs_1 $[t, t']\mathsf{Press} \rightarrow I((t, t']\mathsf{pressed})$
dep_1 $C_T([t]\mathsf{pressed} \wedge \neg\mathsf{tick}) \rightarrow R([t+1]\mathsf{tick})$
dep_2 $C_T([t]\mathsf{tick}) \wedge [t, t+29]\mathsf{tick} \rightarrow R([t+29]\mathsf{color} \hat{=} \mathsf{green})$
dep_3 $C_T([t]\mathsf{tick}) \wedge [t, t+58]\mathsf{tick} \rightarrow R([t+59]\mathsf{color} \hat{=} \mathsf{red} \wedge \neg\mathsf{tick})$
obs_1 $[0]\mathsf{color} \hat{=} \mathsf{red} \wedge \neg\mathsf{tick}$
occ_1 $[10, 14]\mathsf{Press}$
occ_2 $[22, 26]\mathsf{Press}$

There is some subtlety in this scenario description. Pressing the button a second time (while tick is true) will not result in a second period of green. Thus, the second Press action (occ_2) is futile. Notice that this represents an interaction between a delayed effect of one action (occ_1) and a second intermediate action (occ_2). In addition, the delayed effect itself (the light is green) has a duration.

6.2 Examples

In the pedestrian light scenario above, we gave an example of a delayed effect that had limited duration and that blocked another effect (the effect of the second pressing of the button). In this section, we investigate some other interesting aspects of delayed effects using some illustrative scenarios.

Interruption and varying delay time. The first of these scenarios illustrates that the delay may be of varying length and that a delayed effect might be interrupted. It involves a fire-cracker which explodes after a delay of between 10 and 15 seconds. There are two features, burn and expl, which are determined by three influences: $\mathsf{burn}^*(\mathsf{T})$ and $\mathsf{burn}^*(\mathsf{F})$ make burn true respectively false, and expl^* causes the cracker to explode. The effects of the actions and dependencies in acs_1, acs_2, dep_4, dep_5 and dep_6 are directed through these influences, and dep_1, dep_2 and dep_3 specify how the influences affect burn and expl, including when they are in conflict. Regarding delays, dep_5 states that if the fuse has been burning for at least 10 time-points, then it might explode, and dep_6 states that the cracker will definitely explode if the fuse has been burning for 15 time-points. This scenario yields the conclusion that there will be an explosion sometimes between time-points 13 and 18.

Scenario Description 6.2

dom_1 $Per(\mathsf{burn}) \wedge Dur(\mathsf{expl},\mathsf{F}) \wedge Dur(\mathsf{burn}^*(v),\mathsf{F}) \wedge Dur(\mathsf{expl}^*,\mathsf{F})$
acs_1 $[t,t']\mathsf{Light} \rightarrow I((t,t']\mathsf{burn}^*(\mathsf{T}))$
acs_2 $[t,t']\mathsf{Extinguish} \rightarrow I((t,t']\mathsf{burn}^*(\mathsf{F}))$
inf_1 $[t](\mathsf{burn}^*(\mathsf{T}) \wedge \neg\mathsf{burn}^*(\mathsf{F})) \rightarrow R([t]\mathsf{burn})$
inf_2 $[t]\mathsf{burn}^*(\mathsf{F}) \rightarrow R([t]\neg\mathsf{burn})$
inf_3 $[t]\mathsf{expl}^* \rightarrow I([t]\mathsf{expl})$
dep_4 $[t]\mathsf{expl} \rightarrow I([t]\mathsf{burn}^*(\mathsf{F}))$
dep_5 $[t,t+9]\mathsf{burn} \rightarrow X([t+10]\mathsf{expl}^*)$
dep_6 $[t,t+14]\mathsf{burn} \rightarrow I([t+15]\mathsf{expl}^*)$
obs_1 $[0]\neg\mathsf{burn}$
occ_1 $[2,3]\mathsf{Light}$

However, adding an Extinguish action will interrupt the burning, according to acs_2 and dep_2.

occ_2 $[5,6]\mathsf{Extinguish}$

Notice that additional action laws that relate to burn can be added incrementally by utilizing burn*, without any need for modifying the underlying description of burn itself. This is one of the strengths of TAL-C.

Concurrent interaction. Concurrent interaction due to delayed effects can also be of a more direct nature, like in the following example with cumulative effects. In order to specify cumulative effects, we need the set theory introduced in the previous chapter on page 97. The following scenario models a bank account, where the balance (the persistent numerical feature balance) can be altered by influences $\mathsf{bal}^+(x)$ with default value 0, where x represents the source (person, bill etc) of the influence. A special function $\mathsf{Bal}^+(t)$ represents the set of non-zero influences $\mathsf{bal}^+(x)$ at each time-point t.

$$in(f, \mathsf{Bal}^+(t)) \equiv \exists x[f = \mathsf{bal}^+(x) \wedge [t]\neg\mathsf{bal}^+(x) \hat{=} 0] \tag{6.3}$$

The values of the members of the set $\mathsf{Bal}^+(t)$ are added together at each time-point t, and this sum is added to the present balance. We also assume that there can be a deficit on the bank account, but that this leads to a remark from the bank. The following scenario describes how an agent (bob) sends a bill to be paid by his bank and then realizes that the balance is too low to cover this bill, and in the last moment manages to make a deposit and avoids getting a remark.

Scenario Description 6.4

dom_1 $Per(\mathsf{balance}) \wedge Dur(\mathsf{bal}^+(x), 0) \wedge Per(\mathsf{amount}(x)) \wedge Per(\mathsf{remark})$
acs_1 $[s,t]\mathsf{Deposit}(a,n) \rightarrow I([t]\mathsf{bal}^+(a) \hat{=} n)$
acs_2 $[s,t]\mathsf{MailBill}(a,b) \wedge [t]\mathsf{amount}(b) \hat{=} n \rightarrow I([t+50]\mathsf{bal}^+(b) \hat{=} -n)$
dep_1 $[t]\mathsf{balance}\hat{=}m \wedge sum(t{+}1, \mathsf{Bal}^+(t{+}1)) = n \rightarrow R([t+1]\mathsf{balance}\hat{=}(m+n))$
dep_2 $C_T([t]\exists n[\mathsf{balance} \hat{=} n \wedge n < 0]) \rightarrow R([t]\mathsf{remark})$
obs_1 $[0]\mathsf{balance}\hat{=}50 \wedge \neg\mathsf{remark}$
obs_2 $[0]\mathsf{amount}(\mathsf{bill1})\hat{=}150$
occ_1 $[2,3]\mathsf{MailBill}(\mathsf{bob}, \mathsf{bill1})$
occ_2 $[52,53]\mathsf{Deposit}(\mathsf{bob}, 100)$

The key here is dep_1, which sums the values of the non-zero $\mathsf{bal}^+(a)$ in $\mathsf{Bal}^+(t)$ and adds this sum to balance. Thus, $\mathsf{Bal}^+(53) = \{\mathsf{bal}^+(\mathsf{bob}), \mathsf{bal}^+(\mathsf{bill1})\}$ where $[53]\mathsf{bal}^+(\mathsf{bob}) \hat{=} 100$ and $[53]\mathsf{bal}^+(\mathsf{bill1}) \hat{=} -150$, which results in $sum(53, \mathsf{Bal}^+(53)) \hat{=} -50$ and $[53]\mathsf{balance} \hat{=} 0$.

Explicit processes. The previous examples have shown how one can abstract away the underlying mechanics of processes. It is, however, important to point out that our approach is well suited for scenarios that model the dynamics of the world in more detail by letting features represent actual physical quantities. For instance, consider somebody filling a jug from a beer cask with a tap. Let the real-valued feature $\mathsf{flow}(x,y)$ represent a flow from x to y, let $\mathsf{Flow}^+(t,y)$ represent the set of non-zero flows into y at t and let $\mathsf{Flow}^-(t,x)$ represent the non-zero flows out of x.

$$\begin{aligned} in(f,\mathsf{Flow}^+(t,y)) &\equiv \exists x[f = \mathsf{flow}(x,y) \wedge [t]\neg\mathsf{flow}(x,y) \hat{=} 0] \\ in(f,\mathsf{Flow}^-(t,x)) &\equiv \exists y[f = \mathsf{flow}(x,y) \wedge [t]\neg\mathsf{flow}(x,y) \hat{=} 0] \end{aligned} \tag{6.5}$$

In the following scenario, dep_2 encodes a difference equation that determines the volume of water in a vessel based on the sum of flows into and out of the vessel. There are also actions for opening and closing the taps (acs_1 and acs_2), and a dependency law that states that a cask with an open tap produces a flow which is a function of the level of liquid inside the cask (dep_1). A_1 is the bottom area of the cask, A_2 is the area of the tap hole and g is the gravitational constant. Initially, the jug is under the tap of the cask (obs_1) and the tap is closed (obs_2).

Scenario Description 6.6

dom_1 $Per(\mathsf{vol}(x)) \wedge Per(\mathsf{open}(x)) \wedge Per(\mathsf{cask}(x)) \wedge Per(\mathsf{under}(x)) \wedge$
$Dur(\mathsf{flow}(x,y),0)$

acs_1 $[s,t]\mathsf{OpenTap}(x) \rightarrow ([s]\mathsf{cask}(x) \rightarrow R((s,t]\mathsf{open}(x)))$

acs_2 $[s,t]\mathsf{CloseTap}(x) \rightarrow ([s]\mathsf{cask}(x) \rightarrow R((s,t]\neg\mathsf{open}(x)))$

dep_1 $[t]\mathsf{vol}(x) \hat{=} v \wedge [t{+}1]\mathsf{cask}(x) \wedge \mathsf{open}(x) \wedge \mathsf{under}(x) \hat{=} y \rightarrow$
$I([t{+}1]\mathsf{flow}(x,y) \hat{=} (A_2\sqrt{2g\tfrac{v}{A_1}}))$

dep_2 $[t]\mathsf{vol}(x)\hat{=}m \wedge sum(t{+}1,\mathsf{Flow}^+(t{+}1,x)) = n \wedge$
$sum(t{+}1,\mathsf{Flow}^-(t{+}1,x)) = k \rightarrow R([t+1]\mathsf{vol}(x)\hat{=}(m{+}n{-}k))$

obs_1 $[0]\mathsf{under}(\mathsf{cask1}) \hat{=} \mathsf{jug1} \wedge \mathsf{cask}(\mathsf{cask1})$

obs_2 $[0]\neg\mathsf{open}(x)$

obs_3 $[0]\mathsf{vol}(\mathsf{cask1}) \hat{=} 100 \wedge \mathsf{vol}(\mathsf{jug1}) \hat{=} 0$

occ_1 $[5,6]\mathsf{OpenTap}(\mathsf{cask1})$

occ_2 $[7,8]\mathsf{CloseTap}(\mathsf{cask1})$

Opening the tap of the cask (occ_1) produces a flow ($\mathsf{flow}(\mathsf{cask1},\mathsf{jug1})$) from the cask to the jug that decreases as the level of beer in the cask decreases, and closing the tap (occ_2) stops the flow. Notice that our representation of concurrency makes it easy to for instance open an additional cask above the jug; the total flow into the jug would then be the sum of flows from the casks.

6.3 Conclusions

We have presented a method for representing delayed effects of actions which utilizes previous results on ramification and concurrency. As a matter of fact, the formalism used here (TAL-C) is identical to the one presented in Chapter 5, but whereas that chapter concentrated on the interaction between more or less immediate effects of actions, here we exploit the potential to deal with the additional dimension of delays. There are essentially three features of TAL-C that provide this potential. First, there is a notion of explicit time that is independent of action, and that allows us to easily formulate temporal expressions such as "after 15 seconds". Second, there are dependency laws that allow us to describe delayed effects outside of action laws, in addition to dealing with complex ramifications. Third, there is support for concurrency, which is based on the aforementioned dependency laws and a distinction between persistent and durational features. This permits the delayed effects of an action to interact with other actions. In fact, it is the complications that concurrency adds that are the fundamentally difficult issue in representing delayed effects, and maybe that explains why so little progress has been made on the subject. The examples have illustrated delayed effects with duration, interactions with other actions including interruptions, and the modeling of actual processes.

Chapter 7

Object-Oriented Reasoning about Action and Change

Traditionally, the action and change community has primarily used toy domains as benchmarks for testing the semantic adequacy of formalisms. Example 2.1.2 in this thesis is an example of this type of domain. Most of the time, action scenarios in the literature can be described in words using a couple of sentences and the logic representation is seldom more than a page long, with the sentences grouped together by type rather than structure. These toy examples are used in order to highlight or explain some particular point the author wants to make and do have scientific value. However, with some of the classical RAC problems at least partially solved and well understood, and with powerful tools available for reasoning about action scenarios, it is now both possible and necessary to model larger, more complex domains if formalisms of the type described in this thesis are to be applied practically.

When we cease modeling toy domains and begin working with more complex examples, it becomes painfully apparent that there is a lack of methodologies and tools for representing large application domains. There are no principles of good form, like the "No Structure in Function" principle from the qualitative reasoning community [16].

The following topics and questions springs to mind when trying to frame the proper context to address structuring issues in RAC:

- **Elaboration tolerance**: How do we ensure that a domain can initially be modeled at a high level, with the possibility to add further details at a later stage without completely redesigning the domain description? How do we design domain descriptions that can be modified in a convenient manner to take account of new phenomena or changed circumstances?

- **Modularity and reusability**: How can particular aspects of a domain be designed as more or less self-contained modules? How do we provide support for reusing modules?
- **Consistency**: How can complex domains be modeled in a consistent and systematic way, to allow multiple designers to work on a domain and to enable others to understand the domain description more easily?

These questions point toward the need for a framework or a methodology that provides the tools for modeling larger domains. One possibility is to try and apply abstraction and structuring methods used in software engineering to what we call "Logic Engineering". The object-oriented paradigm is one such structuring method that we will focus on. The object-oriented paradigm (for example [2, 11]) claims to provide a more direct mapping to the way we think about reality in an intuitive manner.

An object is an encapsulated *abstraction* of some part of reality that offers specific services to the surrounding world. These services are called *methods* and are the only way to interact with objects. The methods are offered to prospective users by means of an *interface*, the actual implementation being hidden from the user. A method is invoked by sending a *message* to the respective object telling it to execute a specific method.

There are certain powerful principles that make object-orientation suitable for modeling larger RAC scenarios.

- *Modularity*, that is, the decomposition of large and complex systems into smaller modules or objects that interact with each other.
- *Classes* of objects can be defined in advance and stored in a library. Each object is created as an *instance* of an already existing class and contains the same features as its class. This facilitates reusing models.
- The concept of reusability becomes even more powerful in combination with *inheritance*. A new class of objects can be easily created as a specialization and/or extension of already existing classes. A subclass inherits the properties of its parents, and often also adds its own properties.

In the classical object-oriented view as applied to programming, a method is a sequence of code that is procedurally executed when the method is invoked. In our approach, however, a method will contain a set of rules that has to be satisfied whenever the method is invoked. This means that we can invoke

methods over intervals of time and that several methods might be invoked concurrently.

In addition to the above standard object-oriented features we add *constraint methods* that contain rules that must always be fulfilled by all instances of a class. They can be viewed as methods that are invoked at all time-points. This allows us to express many RAC constructions, for example state constraints, but still retain an object-oriented viewpoint. A constraint method can in some senses be compared to an invariant.

7.1 Modeling Object-Orientation in TAL-C

As we have shown previously in this thesis, TAL-C is a flexible and fine-grained language suitable for handling a wide class of domains. The intention of this chapter is to show how to use a number of different aspects of object-orientation in the TAL-C language as a structuring mechanism for domain descriptions, thereby supporting the modeling of more complex and larger domains and the reuse of parts of old domain descriptions when modeling related domains.

The object-orientation can be represented directly in the TAL-C surface language $\mathcal{L}$(ND). The versatility of durational fluents makes it straightforward to model many of the aspects of the object-oriented paradigm that is hard to model in a monotonic logic. Although some of the constructions may initially seem cumbersome, it is possible to introduce a new set of macros in $\mathcal{L}$(ND) to hide them. This form of abstraction already is an integral part of TAL and its methodology.

We use the same logic as in the previous chapters, but with a slight addition for defining the fluent value domains used to simplify the presentation. The standard manner of defining finite domains is by directly listing its elements. An example of this is the following definition of the boolean domain:

dom `boolean` $= \{\top, \bot\}$

The new approach to defining domains is to first describe the relation between the domains and then to add the elements. An example of this is:

dom THING
dom VEHICLE *extends* THING
dom CAR *extends* VEHICLE
obj **agent1**: THING
obj **helicopter1**: VEHICLE
obj **volvo1**: CAR

From this it is possible to create the closure of the domains. In the example, the THING domain is $\{\mathbf{agent1}, \mathbf{helicopter1}, \mathbf{volvo1}\}$, the VEHICLE domain is $\{\mathbf{helicopter1}, \mathbf{volvo1}\}$ and the CAR domain contains only **volvo1**.

This way of representation means that objects cannot be constructed and terminated dynamically. All objects exists at all time-points.

In this chapter we demonstrate the technique using a small application involving a watertank. We begin with two classes called TANK and FLOWTANK. The only interaction possible with the TANK class is to set the volume in the tank to a specified number. The FLOWTANK class has the same behavior as the TANK class but in addition it is possible to add a flow into or out of the tank. We emphasize that the approach targets larger, more complex application domains. Chapter 9 provides three larger scenario descriptions where these ideas are exemplified.

7.1.1 Classes

The basic idea behind our approach is to model classes as sets, and instantiated objects as elements of these sets. Since TAL-C is an order-sorted logic, the existing mechanisms for handling the fluent value domains can be used for our purpose.

In our watertank example, the tank class with the instantiated object **tank1** would be represented as a domain called TANK containing the element **tank1**. The domain definition in $\mathcal{L}$(ND) looks like this:

dom TANK *extends* OBJECT
obj **tank1**: TANK

An object is a member of a class if and only if it is defined as an instance of the class itself or of its subclasses.

This technique ensures that inheritance can be handled in a straightforward manner. If class B extends class A, then B is a subset of A. This means that it is possible to quantify over all objects of a given class, which will be necessary when defining methods. For our watertanks example we would add the following definitions:

dom FLOWTANK *extends* TANK
obj **tank2**: FLOWTANK

The resulting closure, that takes place at translation time, is that FLOWTANK = $\{\mathbf{tank2}\}$ and TANK = OBJECT = $\{\mathbf{tank1}, \mathbf{tank2}\}$. At translation time we also mechanically construct a domain called `classnames` that contains the names of the classes and a *subclass* fluent representing the class structure. The fluent $subclass(c_1, c_2)$ is true if c_1 is a subclass of c_2.

In our example, the *subclass* relation would be as follows:

$$\begin{array}{ll}\textsf{dom} & \texttt{classnames} = \{\text{OBJECT}, \text{TANK}, \text{FLOWTANK}\} \\ \textsf{acc} & \forall t, c_1 \in \texttt{classnames}, c_2 \in \texttt{classnames} \\ & \quad [t]\mathit{subclass}(c_1, c_2) \leftrightarrow \\ & \quad\quad ((c_1 = \text{FLOWTANK} \wedge c_2 = \text{OBJECT}) \vee \\ & \quad\quad\ (c_1 = \text{FLOWTANK} \wedge c_2 = \text{TANK}) \vee \\ & \quad\quad\ (c_1 = \text{TANK} \wedge c_2 = \text{OBJECT}))\end{array}$$

Attributes in an object

The attributes in a class are modeled as standard TAL-C fluents. Since each object of a class should have its own copy of each attribute, all attribute fluents take a single argument of the same type as the class where it was defined.

For example, our TANK class needs a *volume* attribute. Assuming we have a floating point value domain `float`, the *volume* attribute can be modeled as a persistent fluent taking a water tank as an argument:

$$\textsf{attr} \quad \mathit{volume}(\text{TANK}) : \texttt{float}$$

Clearly, since FLOWTANK is a subsort of TANK, any FLOWTANK will also have a volume. In other words, the attribute is automatically inherited by subclasses of TANK taking a flow tank as argument:

The FLOWTANK class also needs a *flow* attribute, which can be modeled as a fluent

$$\textsf{attr} \quad \mathit{flow}(\text{FLOWTANK}) : \texttt{float}$$

Any object of a subclass of FLOWTANK will have a *flow* attribute.

Methods

The only legal interaction between objects is by method invocations. We define three types of methods: procedures, functions and constraint methods. Procedures are used to change the internal state of an object and have no return values, functions do not cause any change but have a return value, and constraint methods represent rules or constraints that are not explicitly invoked but must hold at all time-points.

Procedures: In our approach, procedure invocations are modeled using invocation fluents. For each class c with the corresponding fluent value domain DOMAIN, and for each procedure m we wish to define in that class with arguments of sorts $\langle s_1, \ldots, s_n \rangle$, we define a durational *invocation fluent*

$m(\text{OBJECT}, \text{DOMAIN}, s_1, \ldots, s_n)$: `boolean` with default value false. At any time-point where an object o wants to invoke this procedure in another object o', with the actual arguments $x_1, \ldots, x_n$, it should make $m(o, o', x_1, \ldots, x_n)$ true. We will often use the form $o'.m(o, x_1, \ldots, x_n)$ as a syntactic sugar for $m(o, o', x_1, \ldots, x_n)$.

Suppose, for example, that the TANK class should have a procedure set-volume(f : `float`). We add a durational fluent set-volume(OBJECT, TANK, `float`) : `boolean` with default value false. Then, the object `user1` can call `tank1`.SET-VOLUME(2.0) at some time-point t by making set-volume(`user1`, `tank1`, 2.0) true at t using an interval reassignment formula.

What remains is to define the set-volume procedure. This is done using a dependency constraint that is triggered whenever the invocation fluent is true for some combination of arguments. The basic structure of the definition looks like this:

dep $\forall t, caller \in \text{OBJECT}, self \in \text{TANK}, f \in$ `float`
$[t]self.\text{set-volume}(caller, f) \rightarrow I([t]volume(self) \hat{=} f)$

This dependency constraint states that if any object *caller* calls the set-volume procedure in the tank *self* with argument f, then the volume in *self* becomes f.

Since all objects created from subclasses of TANK by necessity are members of the TANK domain, this method can be invoked on all of them. The above example models a public procedure, which means that any other object can invoke it. If we want to make the method protected, which means that it can only be invoked by objects belonging to TANK or one of its subclasses, the domain of *caller* can simply be set to TANK instead of OBJECT. Note that the *caller* variable can be ignored if the issue of method privacy is irrelevant in the domain.

Constraint methods: In contrast to normal object-oriented programming, some types of behavior have to be active at all time-points. For this we introduce a special type of methods that we call constraint methods. A constraint method looks like a procedure with the exception that we do not require any invocation fluent to be true in order for the method to be active. An example of this is the flow in a flowtank. The changing of the volume does not depend on any method invocation; it should automatically be done at every time-point. The constraint method for this would look like the following:

dep $\forall t, self \in \text{FLOWTANK}, f_1, f_2 \in$ `float`
$[t]flow(self) \hat{=} f_1 \wedge [t]volume(self) \hat{=} f_2 \rightarrow$
$I([t+1]self.\text{set-volume}(self, f_1 + f_2))]$,

where *flow* is the inflow of the tank minus the outflow of the tank. This constraint means that if at time-point t we have flow f_1 and volume f_2 then we invoke the set-volume method with argument $f_1 + f_2$ at time-point $t + 1$.

Functions: Functions are a special case of a constraint method. Functions are used to get values from objects. Instead of representing the invocation with a boolean fluent, as we do for the procedures, we let a *return value fluent* have the same domain as the return value, as in:

dep $\forall t, caller \in$ OBJECT, $self \in$ TANK
$I([t]self.\text{query-volume}(caller) \hat{=} \texttt{volume}(self))$

7.1.2 Elaborating a Class

Since class definitions are not monolithic, an existing class can easily be extended with new attributes and methods without the need to modify the old class definition. It is also possible to create subclasses that have additional attributes or methods. Finally, method implementations in a superclass can be overridden (redefined) in a subclass.

Overriding Method Implementations

A useful feature of object-orientation is the ability to override methods: A method defined in a superclass may be redefined in a subclass, and the new definition takes precedence over the old definition. Say, for example, that we later want to use the FLOWTANK class but we want to have an upper limit on the amount that we can put into the tank (for example, 10 units of water). In this case we construct a new class called OFTANK (OverFlow Tank) to represent such tanks. This class extends the FLOWTANK class with a new method description for the set-volume method that overrides the old behavior. For this to work in our approach, two things have to be done when a method is defined. First, we need to have a way of blocking a method from being invoked. Secondly, when we define a method, we should block all methods with the same name in the superclasses. To do these things we introduce a durational fluent called *override*(`object`,`method`,`classname`) which normally is false, to represent that for a given object `object`, the method `method` defined in class `classname` is overridden.

The first step is made by adding a statement of the following form each time a method is defined:

dep $\forall t, c \in \texttt{classnames}, i \in$ CURRENTCLASS (7.5)
$[t]subclass(\text{CURRENTCLASS}, c) \rightarrow I([t]override(i, \text{methodname}, c)),$

where CURRENTCLASS is the class in which the method is being defined, methodname is the name of the method being defined, and i ranges over all instances of class CURRENTCLASS. The intended meaning of the above statement is that *override* should be true for all methods with the same name, defined in superclasses. For notational convenience we will use the macro **ClassMethod**(CURRENTCLASS, methodname) as a shorthand for statements of type (7.5).

The second step consists of adding, in our method definitions, the requirement that the method is not overridden for the given object in the current class. We will use the macro **Invoked**(CURRENTCLASS, methodname, $\overline{f}$) as a shorthand[1] for

$[t]$methodname$(caller, self, \overline{f}) \wedge$
$[t]\neg override(self,$ methodname, CURRENTCLASS$)$.

The set-volume method we defined earlier should instead be written in the following way, to accommodate the possibility of overriding:

dep_{1a}	**ClassMethod**(TANK, set-volume)
dep_{1b}	$\forall t, caller \in$ OBJECT, $self \in$ TANK, $f \in$ `float`
	Invoked(TANK, set-volume, f) $\rightarrow I([t]volume(i) \hat{=} f)$

If a method is defined as above, it overrides all methods with the same name higher in the hierarchy. It also makes it possible to override this method implementation if some subclass redefines the method.

Example 7.1.1
The OFTANK can now be modeled in the following way:

dom	OFTANK *extends* FLOWTANK
attr	*overflow*(OFTANK) : `boolean`
dep_{2a}	**ClassMethod**(OFTANK, set-volume)
dep_{2b}	$\forall t, caller \in$ OBJECT, $self \in$ OFTANK, $f_1 \in$ `float`
	Invoked(OFTANK, set-volume, f_1) $\rightarrow$
	$I([t]volume(self) \hat{=} min(f_1, 10))$
dep_{2c}	$\forall t, caller \in$ OBJECT, $self \in$ OFTANK, $f_1 \in$ `float`
	Invoked(OFTANK, set-volume, f_1) $\wedge f_1 > 10 \rightarrow I([t]overflow(self))$

We see that the overriding is taken care of automatically. □

Example 7.1.2
In the classical penguin example BIRDs can in general fly, but PENGUINs cannot. The fact that an object can fly is represented by the query-flies

[1] Note that the **Invoked** macro is context dependent on *self* and *caller*.

function. The PENGUIN class extends the BIRD class but overrides the query-flies function with a new function that always returns false.

dom	BIRD *extends* OBJECT
dep_{1a}	**ClassMethod**(BIRD, query-flies)
dep_{1b}	$\forall t, self \in$ BIRD $I([t]self.\text{query-flies}() \mathrel{\hat{=}} \top)$
dom	PENGUIN *extends* BIRD
dep_{2a}	**ClassMethod**(PENGUIN, query-flies)
dep_{2b}	$\forall t, self \in$ PENGUIN $I([t]self.\text{query-flies}() \mathrel{\hat{=}} \bot)$
obj	**tweety** : BIRD
obj	**opus** : PENGUIN

Here query-flies(**tweety**) is true and query-flies(**opus**) is false at all time-points. We have ignored the *caller* variable in this example since it seems farfetched to apply method privacy issues here. □

7.2 Elaboration Tolerance

According to McCarthy [75], elaboration tolerance is the ability to accept changes to a persons or a computer programs representation of facts about a subject without having to start all over. Several of the ideas used in the object-oriented paradigm make it easier to build elaboration tolerant scenarios. This is not surprising since many characteristics of the object-oriented paradigm such as modularization and the possibility to reuse code support elaboration tolerant programming to some extent.

With inheritance it is possible to specialize a class, adding more methods and constraints. Overriding is another aspect that is useful for increasing elaboration tolerance. It allows us to change some behaviors of a class without having to know all the details of that class. This way we do not have to apply any "surgery"[2] if we desire to change the behavior of a subclass. We only have to override the methods that we want to change, leaving the original domain description unchanged.

An example of application of these techniques can be found in Section 9.4, where the well-known missionaries and cannibals scenario is gradually elaborated using these mechanisms as a basis.

[2]McCarthy uses the term surgery to describe the act of going through the scenario description and changing specific values by hand.

7.3 Related Work

Much work has been done in applying object-oriented concept to the application of knowledge representation. One such area is description logics (see for example Brachman et al. [14] and Borgida et al. [12]). Description logics are languages tailored for expressing knowledge about concepts (similar to classes) and concept hierarchies. They are usually given a Tarski style declarative semantics, which allows them to be viewed as sub-languages of predicate logic. One starts with primitive concepts and roles, and can use the language constructs (such as intersection, union and role quantification) to define new concepts and roles. The main reasoning tasks are classification and subsumption checking. Due to this, description logic hierarchies are very dynamic and that it is possible to add new concepts or objects at runtime that are automatically sorted into the correct place in the concept hierarchy. Some work has been done in combining description logics and reasoning about action and change (see for example Artale and Franconi [6]).

The modeling methodology presented in this chapter has a very simple class hierarchy that is constructed at translation time and is thereafter static. Classes have to be explicitly positioned in the hierarchy and classes and objects cannot be constructed once the narrative has been translated. Description logics do not represent class methods or explicit time, both of which are essential in the work presented here.

The approach presented in this chapter bears more resemblance to object-oriented programming languages such as Prolog++ [79], C++ or Java. In these languages, a method is a sequence of code that is procedurally executed when the method is invoked. In our approach, however, a method is a set of rules that have to be satisfied whenever the method is invoked. The fact that delays can be modeled in TAL means that methods can be invoked over intervals of time and that complex processes can be modeled using methods. It is also possible to invoke multiple methods concurrently.

An interesting approach to combining logic and object-orientation is Amir's object-oriented first-order logic [5], which allows a theory to be constructed as a graph of smaller theories. Each subtheory communicates with the other via interface vocabularies. The algorithms for the object-oriented first-order logic suggest that the added structure of object-orientation can be used to significantly increase the speed of theorem proving.

The work by Morgenstern [78] illustrates how inheritance hierarchies can be used to work with industrial sized applications. Well-formed formulas are attached to nodes in an inheritance hierarchy and the system is applied

to business rules in the medical insurance domain. A special mechanism is used to construct the maximally consistent subset of formulas for each node, given its inheritance. Our inheritance in TAL is much simpler since multiple inheritance is prohibited. Applicability of our methods is solely based on the name of the method, not by using a sophisticated mechanisms such as in Morgenstern's work.

7.4 Conclusions

The need for a methodology for scenario description construction becomes evident as soon as we leave the area of toy examples and begin to model realistic dynamic domains. As the size and complexity of domains increase, it becomes more and more difficult to read, modify and debug the domain descriptions.

This chapter has presented a way to apply object-oriented modeling to an already existing logic of action and change, TAL.

The advantage of the work presented here is that larger domains can be modeled in a more systematic manner and that we can group rules together in such a way that it is possible to locally change the representation in a meaningful way. This leads to increased reusability and elaboration tolerance.

The main difference between our work and other approaches to combining knowledge representation and object-orientation is due to the explicit time-line in TAL. Methods can be called over time periods or instantaneously, concurrently or with overlapping time intervals. Methods can relate to one state only or describe processes that take many time-points to complete.

Our approach also allows representation of phenomena not usually modeled in sequential object-oriented languages. It is for example straightforward to sum the arguments of multiple method invocations taking place concurrently. An example of this would be to say that we allow any number of objects to set their individual flow into a tank at the same time. The resulting net flow is the sum of the flows from each tank, that is, the sum of the arguments of all invocations of set_flow.

It should be emphasized that the ideas presented in this chapter do not require any modification of the TAL-C language or semantics, only a restriction on the use of the surface language. In this manner, we enforce more structure on our narratives in order to attain modularity and reusability. It is also likely that these techniques can be used to make theorem proving in $\mathcal{L}$(FL) more efficient, although this has not been the focus of our work.

The work by Amir on an object-oriented first-order logic [5] discussed in the previous section supports this argument.

Finally the modularization also provides a nice interface for hybrid narratives in the sense that some of our objects do not necessarily have to be encoded in our logic. For example, we can define a class for doing complex mathematics as an outside source, implemented procedurally in another object-oriented programming language. The interaction between the TAL classes and the semantic attachments (the outside classes) would be handled in a manner similar to remote method invocations (RMI) in Java [1].

Chapter 8

Using TAL for Control

There are several high-level cognitive tasks for which a logic of action and change can be used such as prediction, postdiction or planning. The logic can be used as a means of specifying and understanding the task, or as an actual basis for implementation. In the previous chapters of this thesis we have dealt with tasks involving a fixed set of actions, specified by the user. The system is then used to determine the set of possible world developments that are consistent with the execution of these actions and with additional knowledge such as domain and dependency constraints — that is, the system has focused on the prediction and postdiction tasks. In addition the focus of the work has been on modeling with less of an emphasis on actual on-line usage in robots or with controllers. Tasks such as on-line planning or control would require our system to work in a different way. Partial narratives would continually be constructed and reasoned about together with actual observations. Given some observations of fluent values, the system has to select appropriate actions to be invoked in a goal-directed manner.

The aim of this section is to study how TAL can be used to construct entities that can actively influence the world model towards a goal state. This way we can use TAL not only as a simulation tool but also to construct components used in goal-directed agents. The role of the scenario constructor changes from writing lists of exact commands to constructing entities able to generate commands interactively.

In Sandewall's Features and Fluents [90], the modeled system is viewed as a game between *ego* and *world*. The ego and the world take turns. The ego invokes actions and the world changes the values of fluents in response to the actions.

We will follow this terminology and use the word *ego* to describe the

entity that chooses what methods to invoke. The observations and the dependency laws describing the environment will be called the *world model* and the layer between ego and world model will be called the *ego-world interface*. Traditionally the ego-world interface consists of action descriptions.

8.1 Other Approaches

Research with logics of action and change often focuses on the properties of actions and how they affect the modeled world. Few of them have been extended as specification or implementation languages for practical problems. One exception is GOLOG [58, 87].

GOLOG is a situation calculus-based[1] logic programming language for defining complex actions using a repertoire of user-specified primitive actions. By using macros it is possible to write robotic control specifications that resemble a computer program that generates sequences of actions that lead towards a specified goal state.

GOLOG provides the usual kinds of imperative programming language control structures as well as three types of nondeterministic choice:

1. Sequence: $\alpha;\beta$ Do action α, followed by action β.
2. Test action: $p?$ Test the truth value of p in the current situation.
3. While loops: **while** p **do** α **endWhile**
4. Conditionals: **if** p **then** α **else** β
5. Nondeterministic action choice: $\alpha|\beta$ Do α or β.
6. Nondeterministic choice of action argument: $(\pi x)\alpha$ Nondeterministically pick a value for x, and for that value of x, do the action α.
7. Nondeterministic repetition: α^* Do α a nondeterministic number of times.
8. Procedures, including recursion.

The semantics of GOLOG programs is defined by macro-expansion using a ternary relation Do, where $Do(\delta, s, s')$ is an abbreviation for a situation calculus formula whose intuitive reading is that s' is a terminating situation of an execution of the complex action δ starting in situation s.

[1] We will assume that the reader is familiar with the situation calculus [57, 76, 74].

Do is defined inductively on the structure of its first argument as follows:

Primitive actions:

$Do(a, s, s') \stackrel{\text{def}}{=} Poss(a, s) \wedge s' = do(a, s)$

Test actions:

$Do(\phi?, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge s' = s$

Sequence:

$Do(\delta_1; \delta_2, s, s') \stackrel{\text{def}}{=} (\exists s^*).(Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s'))$

Nondeterministic choice of two actions:

$Do(\delta_1 | \delta_2, s, s') \stackrel{\text{def}}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$

Nondeterministic choice of action arguments:

$Do((\pi x)\delta(x), s, s') \stackrel{\text{def}}{=} (\exists x) Do(\delta(x), s, s')$

Conditionals:

if p **then** α **else** $\beta \stackrel{\text{def}}{=} (p?; \alpha) | (\neg p?; \beta)$

Similar (but more complicated) definitions are given for iteration and procedures.

Several different flavors of GOLOG exist. The one described here is due to Reiter [87]. A more detailed description can be found in Levesque et al. [58]. Further extensions of GOLOG is an active area of research.

8.2 General Framework

The approach developed in this chapter is inspired by GOLOG, but instead of creating a new control language, we begin by providing a very general framework for the ego-world interface, where the nature of the ego is left unspecified. We then show some examples of how egos can be constructed in TAL, using control structures that are similar to common programming commands.

Traditionally the ego-world interface has consisted of action descriptions, stating how the fluents that represent a domain are affected by the invocation of a specific action.

We suggest a generalization taking advantage of the object-oriented ideas proposed earlier in this thesis. All interaction between objects is handled via method invocations, and since the world consists of objects all interaction between ego and world model also *has* to take place through method calls. This changes the focus away from the actions towards interaction between objects. In our examples, the actions are only used for external intervention in the world, such as initialization.

The world model consists of a number of TAL objects.

The ego-world interface consists of the methods provided by the world that the ego can use to manipulate and interact with the world.

The ego is only allowed to interact with the world through the ego-world interface, using function methods as sensors and procedure methods as actuators. The exact nature of the ego may vary between different applications. For example, it can consist of a list of action invocations, a TAL object, or an external program interacting with TAL such as a theorem prover or model generator.

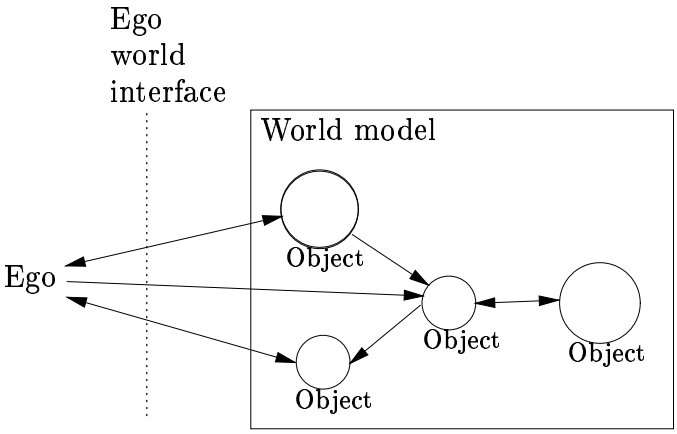

Figure 8.1: The ego–world interaction.

The modular separation between world model and ego makes it possible to use different egos for different control aspects of a problem that one is interested in solving. A world model can have one ego controlling one part of the system and another ego for a different part. These egos should only interact indirectly, via the world model.

Consider an example with a lift and people using the lift. There could be one ego controlling the lift and one ego for each person in the model. Clearly the lift ego uses a different part of the ego-world interface than the person egos do.

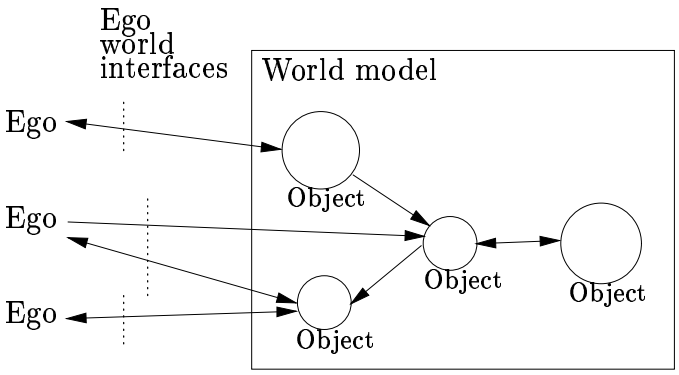

Figure 8.2: Multiple egos.

8.3 Control Objects in TAL

The development of TAL presented in this thesis has concentrated on the world model rather than the ego. Here we propose a way of taking advantage of the tools developed for the world model in the construction of sophisticated egos. Again, we do not have to introduce any new constructs in the logical language associated with TAL. We do modify the surface language with macros and define their translations to the base logic.

In the following, assume the existence of an object-oriented world model that describes how fluents relate to each other and an ego-world interface describing the possible interactions between ego and world. There are then several possible ways of constructing a system that plans how to progress from a (possibly incomplete) initial state to a (possibly incomplete) goal state. Clearly a complete search over all combinations of action invocations is infeasible in all but the smallest cases, although conceptually, this is the search space at hand.

The most natural suggestion is to add our representation to a planner and let it compute a plan. Unfortunately most planners lack the expressivity of RAC languages with for example incompletely specified states, nondeterminism, durations, delays and complex concurrency. Work with TALplanner by Kvarnström, Doherty and Haslum [23, 53, 54, 24] investigates the use of TAL for the development of domain dependent planning techniques.

Another possibility is to develop a control language like GOLOG, where it is possible to encode knowledge of how to behave in order to reach the goal state.

The approach studied in this chapter is inspired by the last suggestion, but instead of developing a new control language like GOLOG, we use the generalization proposed in Section 8.2 and provide examples showing how egos can be written in TAL. We will also illustrate how some control constructs can be trivially represented with this approach while others are much harder to model.

The advantage of constructing egos in TAL is that some of the more useful representational concepts are acquired for free, such as nondeterminism, side-effects and concurrency.

The general framework presented in this chapter allows great freedom in deciding how to construct an ego. The most elementary type of ego is simply one that uses a list of method calls and specifies the time-points at which the methods should be called, similar to a TAL narrative. A slightly more dynamic type of ego could be built using a list of case statements. Such an ego can easily be implemented using dependency laws for each case, with function methods in the precondition and procedure methods as effects for each programming instantiation. These dependency laws can either be totally concurrent, which means several procedure calls can be invoked by the ego at the same time-point, or they could be prioritized, where only the first applicable dependency law is triggered. A prioritized list of dependency laws could be implemented in TAL as follows:

dep_1 $[t]\mathbf{obj}_1.\mathsf{query\_too\_high}() \rightarrow I([t]\mathbf{obj}_1.\mathsf{decrease}() \wedge \mathit{triggered} = 1)$

dep_2 $[t]\mathit{triggered} > 1 \wedge [t]\mathbf{obj}_1.\mathsf{query\_too\_low}() \rightarrow$
$I([t]\mathbf{obj}_1.\mathsf{increase}() \wedge \mathit{triggered} = 2)$

dep_3 $[t]\mathit{triggered} > 2 \wedge [t]\mathbf{obj}_1.\mathsf{query\_too\_fast}() \rightarrow$
$I([t]\mathbf{obj}_1.\mathsf{accelerate}() \wedge \mathit{triggered} = 3)$

$\vdots$

If the first dependency law can be triggered, then *triggered* is set to 1. None of the other dependency laws can be triggered, since they all have the requirement that *triggered* must be greater than 1 in the precondition. If the precondition of the first dependency law is false but the second dependency law can be triggered, then *triggered* is set to 2 and none of the following laws can be applied, and so on.

If a more complex ego is desired, more abstract modeling tools are beneficial. Classical programming constructs such as loops and if-then-else statements greatly simplifies the construction of the controlling entity. In the following subsection we will describe a number of programming language constructs which can be used in the construction of specific ego programs.

8.4 Control Structures in TAL

To be able to build controllers in TAL, we need some of the common programming language constructs. Below is an outline of how some such constructs can be modeled in TAL.

8.4.1 For loops

Loops are used to do something repeatedly. In C++ for example, the statement "`for (x=1;x<4;x++) obj.func(x);`" means that we first call the method called `func()` in object `obj` with argument 1 then with 2 and finally with 3. In TAL we can specify these kinds of loops both sequentially, as in C++, with one call per time-point, or in parallel, where we call **obj**.func(1), **obj**.func(2) and **obj**.func(3) at the same time. We will call the former a sequential loop and the latter a parallel loop.

Sequential loops The following loop construction will start a loop whenever a trigger fluent becomes true. Starting the loop will make the fluent varname increase its value at each time-point starting with value lb and continuing up to but not including the upper bound ub assuming loopname stays true. Here loopname is a boolean fluent that is true if the loop is running. Different loopnames are required to distinguish between different loops.

dep_1 $[t]C_T(\mathsf{trigger}) \wedge \neg\mathsf{loopname} \rightarrow I([t+1]\mathsf{varname} \mathrel{\hat{=}} lb \wedge \mathsf{loopname} \mathrel{\hat{=}} \top)$

dep_2 $[t]\mathsf{loopname} \wedge (\mathsf{varname} \geq lb) \wedge (\mathsf{varname} < ub) \wedge (\mathsf{varname} = x) \rightarrow$
$I([t+1]\mathsf{varname} \mathrel{\hat{=}} x+1)$

dep_3 $[t]\mathsf{varname} \geq ub \rightarrow I([t]\neg\mathsf{loopname})$

dep_4 $[t]\mathsf{loopname} \rightarrow I([t]\mathbf{obj}.\mathsf{func}(\mathsf{varname}))$

Parallel loops The following loop construction will make **obj**.func(x) true for all $lb \leq x < ub$, whenever trigger is true.

dep_1 $\forall x.[t]\mathsf{trigger} \wedge x \geq lb \wedge x < ub \rightarrow I([t]\mathbf{obj}.\mathsf{func}(x))$

This type of loop is executed whenever trigger is true, while the sequential loop is *started* whenever trigger *becomes* true and runs over an interval of time.

8.4.2 While loops

Some aspects of the while-statement "`while (condition) obj.func();`" can be modeled through an ordinary dependency law.

dep_1 $[t_1, t_2]\mathsf{condition} \rightarrow I([t_1, t_2]\mathbf{obj}.\mathsf{func}())$

This does not capture the whole complexity of while-statements. Our statement is reactivated each time the condition is true. To avoid this we could either introduce a fluent that represents that the while-statement is ended, or we could use a program counter as we will describe in Section 8.4.4.

8.4.3 If then else

An "`if (condition) obj1.func1(); else obj2.func2();`" statement is simply translated to

dep_1 $[t]\mathsf{condition} \rightarrow I([t]\mathbf{obj}_1.\mathsf{func}_1())$
dep_2 $[t]\neg\mathsf{condition} \rightarrow I([t]\mathbf{obj}_2.\mathsf{func}_2())$.

Note that these dependency laws are always active as opposed to a normal programming language, where we execute the check once and then move on to the next line of code. Doing things sequentially is more difficult in TAL.

8.4.4 Sequence

Representing sequences is the area where TAL differs the most from ordinary programming languages or control languages such as GOLOG.

One of the most basic assumptions in most programming languages is that commands are executed sequentially: Unless the user explicitly creates multiple threads of execution, the computer finishes executing one command before moving on to the next. This construction suits a broad set of applications where one can read the program as a sort of story, an outline of what will happen.

TAL, on the other hand, is inherently parallel instead of sequential. Thus, whereas most programming languages must specify in some detail how parallel programs should be written, we will now specify in some detail how TAL controllers can be written to handle three slightly different kinds of sequentiality.

Sequences in the sense that one thing happens at time-point x, the next at $x + c_1$ and the next at $x + c_1 + c_2$ can of course be modeled very easily, given that the constants c_1 and c_2 are known in advance. In fact, this type of sequentiality can be modeled more easily than in an ordinary programming languages, where explicit delays would be necessary in order to ensure that the second command is executed c_1 units of time after the first command.

It would probably be more common to model a form of sequentiality where one command A is executed at time x and the next command B is executed

whenever A is finished. This would be slightly more difficult, since A may contain loops and similar constructs. Two cases can be considered.

It may be the case that A is intended to achieve some specific condition ϕ in the world, and that A is finished exactly when ϕ is achieved. In this case, B can simply be triggered by ϕ becoming true – that is, by the condition $[t]\ C_T(\phi)$.

If this is not the case, or if the condition ϕ is too difficult to model or cannot be modeled due to the nature of A, then one must most likely resort to introducing additional "help fluents" to model true sequential behavior. For example, it is possible to introduce an artificial "program counter" to control the execution of sequential commands. This is one of the cases which there is no truly natural way to model in TAL.

Below are three examples of these common constructs for sequentiality.

Example 8.4.1 (Known delay)
When turning the ignition key in the car, the start engine will rotate. After a known interval of time, the car drives away.

dep_1 $[t]\mathbf{car}_1.\mathsf{query\_ignition\_key}() \rightarrow$
$I([t+1]\mathbf{car}_1.\mathsf{start\_engine}()) \wedge I([t+3]\mathbf{car}_1.\mathsf{drive}())$ □

Example 8.4.2 (Delay via world observations)
When turning the ignition key in the car, the start engine will rotate – that is, we call the car's start_engine method, which will eventually start the engine for us. We can detect when this is done using the query_main_engine_running method. When the engine is running, we can drive away.

dep_1 $[t]\mathbf{car}_1.\mathsf{query\_ignition\_key}() \rightarrow I([t+1]\mathbf{car}_1.\mathsf{start\_engine}())$
dep_2 $[t]\mathbf{car}_1.\mathsf{query\_main\_engine\_running}() \rightarrow I([t+1]\mathbf{car}_1.\mathsf{drive}())$

This assumes that there are some rules in the car object that are triggered by $\mathsf{start\_engine}(\mathbf{car}_1)$ and eventually make $\mathsf{query\_main\_engine\_running}(\mathbf{car}_1)$ true. For example, the following rules might suffice:

dep_3 $[t]\mathbf{car}_1.\mathsf{start\_engine}() \rightarrow I([t+c]\mathbf{car}_1.\mathsf{engine\_running}())$
dep_4 $[t]\mathbf{car}_1.\mathsf{engine\_running}() \rightarrow I([t]\mathbf{car}_1.\mathsf{query\_main\_engine\_running}())$ □

Example 8.4.3 (Sequentiality using a Program Counter)
The following part of a program:

```
if (!driver1.carries(key1)) then driver1.get(key1);
while (!car1.running()) driver1.turnkey(car1);
car1.drive();
```

could be translated into the following set of dependency laws:

dep_1 $[t]pc = 1 \wedge [t]\neg\mathbf{driver}_1.\mathsf{carries}(\mathbf{key}_1) \rightarrow I([t+1]\mathbf{driver}_1.\mathsf{get}(\mathbf{key}_1) \wedge pc = 2)$

dep_2 $[t]pc = 2 \wedge [t]\neg\mathbf{car}_1.\mathsf{running}() \rightarrow I([t+1]\mathbf{driver}_1.\mathsf{turnkey}(\mathbf{car}_1))$

dep_3 $[t]pc = 2 \wedge C_t([t+1]\mathbf{car}_1.\mathsf{running}()) \rightarrow I([t+1]pc = 3)$

dep_4 $[t]pc = 3 \rightarrow I([t+1]\mathbf{car}_1.\mathsf{drive}() \wedge pc = 4)$

This shows that sequentiality can be modeled by handcoding the fluent pc that simulates a program counter. This is too cumbersome to be practical for larger programs and indicates one of the weaknesses with this approach.

Macros can be constructed to hide this type of program counters, but this does not remove the problem. □

8.5 Simulation and Planning

A world definition and an ego designed using the control structures discussed above can easily be combined into a single $\mathcal{L}$(ND) narrative. Translating this narrative into the base logic $\mathcal{L}$(FL) results in a theory whose models correspond to the possible world developments that could arise given this world and ego and the initial conditions specified in the narrative. Non-determinism or incomplete information about any part of the world or ego may give rise to multiple models, each of which corresponds to one specific development. A theorem prover can then be used to answer queries, or a model generator such as VITAL can be used to find and visualize all the world developments. In essence, the logic is used to *simulate* both the actions that would be taken by a controller and the possible resulting reactions of the world.

It is also straight-forward to automatically extract the information about the decisions made by the ego and to view this as a *plan* to be executed by an actual physical entity such as a robot. In the presence of non-determinism and incomplete information about the world, this could result in a *conditional* plan where the next action to be invoked by the robot is determined by the actual outcome of its actions at the previous time step. As long as the description of the world is correct and does not exclude developments that could possibly take place, the robot will never end up in a world state not described by the conditional plan. Naturally, the efficiency of this form of planning depends on the efficiency of the theorem proving or model generation mechanism being used.

However, in some cases, it is natural to also specify a goal condition in the form of a constraint on a final state, in addition to, or instead of, the reactive controller of the kind discussed in this chapter. In the missionaries

and cannibals domain discussed in Section 9.4, for example, the goal is that all missionaries and cannibals should end up at the opposite side of the river. This goal is quite difficult to encode into a controller in such a way that any legal world development will end in a state where the goal is achieved.

This kind of goal can be handled at a meta-logical level using standard planning mechanisms, where one searches for a (possibly conditional) plan that will definitely satisfy the state goal. Given certain restrictions, it is also possible to generate plans for this kind of goal simply by stating (using an observation statement) that the goal *is* satisfied at the final time-point. This requires that all non-determinism and incomplete information in the combined narrative (containing ego and world descriptions) corresponds to choices that can be made by the ego or egos, such as whether to cross the river at this time-point or not; claiming that the goal is definitely satisfied will simply restrict the possible choices. The world, on the other hand, must be completely described and deterministic. Otherwise, our claim that the goal will definitely be satisfied might constrain the world itself, as demonstrated in the following example.

Example 8.5.1 (World nondeterminism and planning)
Consider a narrative where the ego has a coin. The ego tosses the coin, and the world determines whether the result is heads or tails. This is represented as non-determinism in the world model. The goal is that the coin will land heads up. If we try to find a plan by simply claiming that the goal will indeed be achieved, there will be a single logical model where the ego tosses the coin and the world has been constrained in such a way that the coin does land heads up. The possible development where the coin lands tails up is not a logical model, since it contradicts the state goal.

Note that if we want to simulate instead of plan, and that the coin landed heads up is viewed as an actual observation rather than a goal, world nondeterminism is unproblematic, and this example would give the right models. □

8.6 Larger Examples

The lift and road network scenarios in Chapter 9 exemplify how the ideas in this section could work in practice.

The lift example shows how a nondeterministic ego can control a lift. The lift controller's ego-world interface consists of the functions query_level and query_pressed and the procedures move_down, move_up and reset_button.

The road network example is more complex. We show how an ego can inherit and extend the behavior of another ego class and how different egos can be attached to different cars. The ego-world interface for a car controlling ego consists of the functions query_place, query_position, query_closest, query_connects, query_green_light and query_distance and the procedures set_place, set_position and move_vehicle.

8.7 Conclusions

Using the proposed framework described in Section 8.2 and the TAL constructs described in Section 8.3 it is possible at least conceptually to design a wide variety of control systems at a high level of abstraction. The main advantages are the high level of modularity and elaboration tolerance and the fact that we avoid the requirement of two separate languages for control and world modeling, thereby gaining the full TAL expressivity in our control objects.

Chapter 9

Examples

One of the main reasons for developing the object-oriented structuring techniques for TAL in Chapter 7 was to be able to model more than just toy domains. In this chapter, we take advantage of the tools developed in Chapter 7 and describe three examples of more complex domains.

- The first example illustrates how a controller, specified in TAL, can be used to govern the behavior of a simple lift.
- The second example shows how inheritance and overriding can be used as powerful tools for elaboration tolerance in the missionaries and cannibals domain.
- The last and most complex example consists of a road network with cars. This domain is based on the example described in the Logic Modelling Workshop [88].

Each example begins with a brief general description followed by explanations of the classes and their methods. Trivial selector or mutator functions will not be described. A trivial mutator is a procedure method and has the following form: [1]

dep_1 **ClassMethod**(PARTICLE, set_acceleration)
dep_2 $[t]\neg\mathsf{override}(\mathit{particle}, \mathsf{set\_acceleration}, \text{PARTICLE}) \wedge$
$\mathit{particle}.\mathsf{set\_acceleration}() = \mathit{Integer} \rightarrow$
$I([t]\mathit{acceleration}(\mathit{particle}) \hat{=} \mathit{Integer})$

[1] This procedure assumes that there can only be one influence on the particle at each time-point. The same technique as used in Chapter 5 can be used to lift that requirement.

A trivial selector function is a function method and typically looks like the following:

dep **ClassMethod**(PARTICLE, query_acceleration)
dep $[t]\neg$override($particle$, query_acceleration, PARTICLE) $\rightarrow$
$I([t]particle.\text{query\_acceleration}() \hat{=} value(t, acceleration(particle)))$

9.1 Implementation and Tests

Section 9.3 presents the lift example and contains detailed explanations of all non-trivial methods. The cannibal and road network example does not explain the methods in detail, the interested reader can study the complete listing in Appendices D, E and F. Those listings have been tested with VITAL, a Java program that implements a significant fragment of TAL. All tests have been run on a Sun Ultra 10 workstation. It should be noted that VITAL does not take advantage of the object-oriented structure of the scenario descriptions. Results by Amir [5] suggests that this could speed up the time to find the solutions considerably.

9.2 Some Terminology

Summation over a number of elements is not part of the TAL logic. It is added to VITAL as a semantic attachment[2] and is useful in a number of instances, for example to avoid some types of conflicts resulting from concurrency.[3]

In method descriptions we will use expressions of the type

$$\sum_{\{g \mid g \in class \wedge [t]g.\text{query\_position}() \hat{=} place\}} value(t, g.\text{query\_size}())$$

to describe summation. In VITAL the same expression would be written as follows:

$\$\text{sum}(\langle group \rangle, [t]\text{query\_position}(group) \hat{=} place, value(t, \text{query\_size}(group)))$

[2]It is also possible to introduce this type of addition by incorporating a minimal portion of set theory into TAL as described in Section 5.6.2.

[3]See the modify_group method in the missionaries and cannibals example in Section 9.4.1 for an example of how summation is used to avoid concurrency conflicts.

9.3 Object-Oriented Modeling I: The Lift Scenario

In this section, we present an example that illustrates how the ideas from the previous chapter can be applied to control a simple lift. The controller should move the lift in such a way that it serves all floors where a lift button has been pressed. This scenario is inspired by a lift example in Levesque et al. [58] with some additional extensions.

9.3.1 Overview of the Design

We have chosen to represent this problem with four classes. The first is the root class:

- OBJECT. The superclass of all other classes. Contains no methods.

There are also two world classes. Their ego-world interfaces are described later.

- BUTTON. Each floor has a button.
- LIFT. The actual lift.

Finally, there is an ego class:

- CONTROLLER. The entity that governs the behavior of the lift via the LIFT's interface.

The basic idea is that the CONTROLLER uses the ego-world interface of the LIFT to serve all floors with pressed buttons. In this example we can press the buttons with an action.

In the scenario described here, one lift, one controller and five buttons will be constructed and initialized. Then we will press the button on floors three and five, through two concurrent invocations of the PressButton action at the initial time-point. The intended solution is that the controller directs the elevator to move to either the third floor and then to the fifth, or first to the fifth floor and then to the third floor.

Button

The BUTTON class extends OBJECT. There should be one BUTTON object per floor. A BUTTON has two attributes: *pressed* (signaling that the button has been pressed) and *attached* (the number of the floor the button is on).

The following methods are associated with button:

- attach_button(*Integer*) sets *attached* to *Integer*. This represents that the button exists on floor number *Integer*.
- query_attached() returns the value of *attached*.
- press_button() sets *pressed* to true which represents that someone has pressed the button.
- reset_button() sets *pressed* to false.
- query_pressed() returns the value of *pressed*.

Lift

The LIFT should contain only the basic functionality of a lift which is that you can make it move up or down. It is the task of another object, the CONTROLLER, to call these methods at suitable time-points. The LIFT has a *currentfloor* and a *topfloor* attribute. The *currentfloor* attribute contains the number of the floor the lift is currently at, and *topfloor* is the number of the highest floor the lift can go to. Possible conflicts with moving the lift up and down at the same time are handled by preferring movement downwards. The following methods are associated with LIFT:

- init_floor(*Integer*) sets *currentfloor* to *Integer*. This procedure should only be used at time-point 0 to set up the scenario.
- query_floor() returns the floor the lift is on.
- set_top_floor(*Integer*) sets the limit on the number of floors the lift serves.
- move_up() moves the lift one floor up if it is currently below the top floor.

$$
\begin{aligned}
&\text{dep} \quad \mathbf{ClassMethod}(\text{LIFT}, \text{move\_up}) \\
&\text{dep} \quad [t]\neg\text{override}(\mathit{lift}, \text{move\_up}, \text{LIFT}) \wedge \\
&\qquad \mathit{lift}.\text{move\_up}() \wedge \\
&\qquad \neg \mathit{lift}.\text{move\_down}() \wedge \\
&\qquad \mathit{lift}.\text{query\_floor}() < \mathit{top\_floor}(\mathit{lift}) \rightarrow \\
&\qquad\quad I([t+1]\mathit{currentfloor}(\mathit{lift}) \mathrel{\hat{=}} \mathit{value}(t, \mathit{currentfloor}(\mathit{lift})) + 1)
\end{aligned}
$$

- move_down() moves the lift one floor down if it is currently above floor 0.

  dep **ClassMethod**(LIFT, move_down)
  dep $[t]\neg$override($lift$, move_down, LIFT) $\wedge$
  $lift$.move_down() $\wedge$
  $lift$.query_floor() $> 0 \rightarrow$
  $I([t+1]currentfloor(lift) \hat{=} value(t, currentfloor(lift)) - 1)$

  It is straightforward to observe that the combination of the two methods move_up and move_down covers all possible cases of overlap of these two procedures. If both procedures are called at the same time-point, the preconditions of move_up will be false which means that only move_down will be active.

Controller

The purpose of the controller is that via the BUTTON's and LIFT's interface it attempts to serve all floors. The CONTROLLER has a *goal_floor* attribute that represents the floor the controller is trying to move the lift to. The *goal_floor* attribute is automatically updated when necessary. Given a *goal_floor*, the controller moves the lift to that floor. The other attribute *controlled_lift* refers to the lift the controller is controlling.

The following two methods are used at setup:

- set_lift(LIFT) sets the lift the controller controls.
- query_controlled_lift() returns the lift the controller controls.

The constraint methods below control the lift via its interface functions:

- new_goal(). If the lift is at the *goal_floor* and at least one button has a request, then nondeterministically choose a new goal floor.

  dep **ClassMethod**(CONTROLLER, new_goal)
  dep $[t]\neg$override($controller$, new_goal, CONTROLLER)$\wedge$
  $lift =$ controlled_lift($controller$)$\wedge$
  $lift$.query_floor() $=$ $goal\_floor(controller)\wedge$
  $\exists button[[t]button$.query_pressed()$] \rightarrow$
  $\exists button[[t]button$.query_pressed()$\wedge$
  $I([t+1]goal\_floor(controller) \hat{=}$
  $value(t, button$.query_attached()))]

- goto_floor(). Make the elevator move towards the goal floor via the move_up or move_down methods in the LIFT object.

  dep **ClassMethod**(CONTROLLER, goto_floor)
  dep $[t]\neg$override($controller$, goto_floor, CONTROLLER)$\wedge$
  $lift = controlled\_lift(controller) \wedge$
  $lift$.query_floor() $< goal\_floor(controller) \rightarrow$
  $I([t]lift$.move_up())

  dep $[t]controlled\_lift(controller) = lift\wedge$
  $lift$.query_floor() $> goal\_floor(controller) \rightarrow$
  $I([t]lift$.move_down())

- serve_floor(). If the controlled lift is at the goal floor, then remove the request in the button at that floor via the buttons reset_button method.

  dep **ClassMethod**(CONTROLLER, serve_floor)
  dep $[t]\neg$override($controller$, serve_floor, CONTROLLER)$\wedge$
  $[t]value(t, controlled\_lift(controller))$.query_floor() $=$
  $goal\_floor(controller)\wedge$
  $[t]button$.query_attached() $= goal\_floor(controller) \rightarrow$
  $I([t]button$.reset_button())

9.3.2 Elaborations

The clear separation between lift, button and controller and the object-oriented modeling makes it trivial to elaborate on the previous design. If it has to be changed to a 20 floor lift, it is *only* the LIFT class that needs to be changed. If the choice of which floor to serve should be modified, all that needs to be done is to let a new controller class extend the controller and override the new_goal method. Modifications like these, but for another scenario, are shown in Section 9.4.

9.3.3 Summary

This domain shows how that TAL can be used to construct egos that actively work towards a goal. In this case the ego's goal is to have served all floors with pressed buttons.

9.4 Object-Oriented Modeling II: The Missionaries and Cannibals Problem

The Missionaries and Cannibals Problem (MCP) is a good example for showing how the object-oriented paradigm discussed in the Chapter 7 can support the construction of elaboration tolerant scenarios.

The basic MCP reads: "Three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross in order to avoid anyone being eaten?" McCarthy [75] illustrates his ideas regarding elaboration tolerance with 19 elaborations of this scenario. Some of these elaborations have been implemented in the Causal Calculator[4] by Lifschitz [65]. The idea behind Lifschitz' approach is to associate each rule with a number. Normally the rule applies, but if we state that the number is abnormal, the rule is ignored. With this mechanism it is possible to incrementally build a system without ever having to go back and change an already existing rule.

Our approach is very similar to Lifschitz' but we propose to take advantage of the functionality developed in Chapter 7 for object-orientation. At the heart it is the same idea, that is to let the rules have an extra precondition that normally is true, but which can be overridden if the user attempts to create a new rule governing the same behavior. In our approach, the old rules are automatically overridden if the scenario constructor creates a subclass that has a method with the same name. As we will see, this makes it easy and natural to modify and extend the basic scenario.

9.4.1 Overview of the Design

The assumption when modeling the basic MCP is that we know it might be extended at a later time-point but we do not know exactly what will be modified.

For the basic scenario we need six classes:

- OBJECT is superclass to all other classes.
- BOAT extends OBJECT and is a class containing all boats.
- GROUP extends OBJECT and is the class of all groups of people

[4] A system for query answering and satisfiability planning designed and implemented at the University of Texas. `http://www.cs.utexas.edu/users/tag/cc/`

- CANGROUP extends GROUP and is the class of all groups of cannibals
- MISGROUP extends GROUP and is the class of all groups of missionaries
- PLACE extends OBJECT. We will use **people_at**(CLASS, *place*) as an abbreviation for

$$\sum_{\{g \mid g \in \text{CLASS} \wedge [t] g.\text{query\_position}() = place\}} value(t, g.\text{query\_size}())$$

  throughout this chapter for determining how many people of a given class CLASS that occupies position *place*.

  The statement **people_at**(GROUP, **onvera**), for example, is the number of people on the boat Vera and **people_at**(CANGROUP, **left**) represents the number of cannibals on the left bank.

- BANK extends PLACE

The VITAL code for the basic scenario is found in Appendix E.2.1.

Object

Every OBJECT class is the superclass of all other classes. The OBJECT has a *position* attribute, representing the location of the object. The *position* attribute is influenced by the following methods:

- set_position(PLACE) Procedure. Sets the position of the object.
- query_position() Function. Returns the position of the object.

Boat

The BOAT class represents the boats that the groups use to cross the river. It has one attribute, *onboard*, that refers to the location onboard the boat. There are three methods in BOAT:

- query_onboard() Function. Returns the PLACE that is onboard the boat.
- move_boat() Constraint method. If there is anybody onboard, then it will move the boat to another (nondeterministically chosen) BANK.
- boat_limit() Constraint method. Governs the limit of passengers on the boat.

Group

The GROUP is the most complex class. It represents all groups of people. It has one attribute, *size*.

- modify_group(GROUP$_2$) Procedure. Setting *group*.modify_group(GROUP$_2$) to x represents that GROUP$_2$ adds x persons to the current group (*group*). Due to the fact that a GROUP can be modified by several objects at the same time, we require modify_group to take an extra argument that is the object that causes the modification of size. The modify_group method computes the sum of all those influences and changes the size accordingly.

  dep **ClassMethod**(GROUP, modify_group)
  dep $[t]\neg\mathsf{override}(group, \mathsf{modify\_group}, \text{GROUP}) \rightarrow$
  $I([t+1]size(group) \mathrel{\hat{=}}$
  $value(t, size(group)) +$
  $\sum_{\{g \mid g \in \text{GROUP}_2\}} value(t, group.\mathsf{modify\_group}(g)))$

- query_size() Function. Returns the number of people in this group.

- move_persons() Constraint method. Nondeterministically move people between GROUPs of the same type (missionaries or cannibals) that are located in connected PLACEs. Note that GROUPs never move – people move by changing the size of two groups.

dep **ClassMethod**(GROUP, move_persons)
dep $[t]\neg\mathsf{override}(group, \mathsf{move\_persons}, \text{GROUP}) \wedge$
$group.\mathsf{query\_type}() \mathrel{\hat{=}} group_2.\mathsf{query\_type}() \wedge$
$[t+1]group.\mathsf{query\_pos}().\mathsf{query\_connection}(group_2.\mathsf{query\_pos}()) \rightarrow$
$\exists Integer\ [-value(t, group_2.\mathsf{query\_size}()) \leq Integer \wedge$
$Integer \leq value(t, group.\mathsf{query\_size}()) \wedge$
$I([t+1]group.\mathsf{modify\_group}(group_2, -Integer)) \wedge$
$I([t+1]group_2.\mathsf{modify\_group}(group, Integer))]$

Cannibals

CANGROUP extends GROUP and adds the following method:

- eat_constraint() Constraint method. Specifies that there cannot be more cannibals than non-cannibals at any place, even if some of them are in a boat.

Missionaries

MISGROUP extends GROUP but adds no new methods.

Place

The PLACE class has an internal variable *connection* that represents the other PLACEs this PLACE is connected to.

- add_connection(PLACE$_2$) Procedure. Makes this PLACE connected to PLACE$_2$
- remove_connection(PLACE$_2$) Procedure. Removes the connection between this PLACE and PLACE$_2$.
- query_connection(PLACE$_2$) Function. Returns true if this PLACE is connected to PLACE$_2$.

Bank

BANK extends PLACE but adds no methods. It merely serves to limit the places the boat can reach, that is, it can only reach BANKs.

General constraints

Due to the nature of our approach we have to add some constraints to limit the search space. Both are due to the fact that we want to remove models where everything stays the same from one time-point to the next. The first constraint states that the size of at least one group has to change, and the second one requires that there is at least one person on the boat except at the first and last time-point.

The actual code is found in Section E.3.

9.4.2 Setting Up the Problem

In order to set up a problem instance, we first have to instantiate some objects. The boat will be called **vera**, there will be two banks (**left** and **right**), and there are groups of missionaries and cannibals in all three places.

obj **left**,**right** : BANK
obj **onvera** : PLACE
obj **vera** : BOAT
obj **cleft**,**cvera**,**cright** : CANGROUP
obj **mleft**,**mvera**,**mright** : MISGROUP

The following observation statements specify the attributes of these objects:

obs $[0]\mathbf{vera}.pos \hat{=} \mathbf{left} \wedge \mathbf{vera}.onboard \hat{=} \mathbf{onvera}$

obs $[0]\mathbf{cleft}.pos \hat{=} \mathbf{left} \wedge \mathbf{cleft}.size \hat{=} 3$

obs $[0]\mathbf{cvera}.pos \hat{=} \mathbf{onvera}$

obs $[0]\mathbf{cright}.pos \hat{=} \mathbf{right}$

obs $[0]\mathbf{mleft}.pos \hat{=} \mathbf{left} \wedge \mathbf{mleft}.size \hat{=} 3$

obs $[0]\mathbf{mvera}.pos \hat{=} \mathbf{onvera}$

obs $[0]\mathbf{mright}.pos \hat{=} \mathbf{right}$

acc $[0]group.size \hat{=} 0 \leftrightarrow (group \neq \mathbf{mleft} \wedge group \neq \mathbf{cleft})$

acc $[0]place_1.connect(place_2) \leftrightarrow ((place_1 = \mathbf{left} \wedge place_2 = \mathbf{onvera}) \vee (place_1 = \mathbf{onvera} \wedge place_2 = \mathbf{left}))$

Finally, a goal is required. We know that the minimal plan length is 12:

obs $[12]\mathbf{mright}.size \hat{=} 3 \wedge \mathbf{cright}.size \hat{=} 3$

9.4.3 Elaborations

We will now use the object-oriented model of the basic MCP domain defined above to show how to model the 19 elaborations defined by McCarthy [75]. We provide timings for some of the elaborations that have been tested in the research tool VITAL [55]. We will also provide some comparisons with the 10 elaborations implemented by Lifschitz [65] in the Causal Calculator [73].[5]

We concentrate on elaboration tolerance for the *domain specification* (the class definitions). Although it would have been possible to model the problem setup in Section 9.4.2 in a defeasible manner using similar techniques, we instead make the assumption that one is generally interested in solving many different problems in the same general domain and that the specific problem instances (such as the number of missionaries and cannibals, the set of river banks, and which banks are connected) are generated from scratch each time. The problem instance definitions for the elaborations below are generally trivial and will usually be omitted.

The original problem

The original problem is solved in 2 seconds by VITAL.

[5]VITAL was run on a 440 MHz UltraSparc machine; the Causal Calculator was run on an unspecified machine. The timing results should not be taken too seriously. No work on complexity have been done with respect to this.

The boat is a rowboat (elaboration 1)

The fact that the boat is a rowboat can be modeled by making **vera** an instance of a new class ROWBOAT that extends BOAT without any additional attributes or methods. The problem is still solved in 2 seconds.

People have hats (elaboration 2)

The missionaries and cannibals have hats, all different. These hats may be exchanged among the missionaries and cannibals.

While missionaries and cannibals used to be interchangeable and could be modeled as groups, they must now be seen as individuals. The following classes and attributes are added:

dom HAT *extends* OBJECT
dom PERSON *extends* OBJECT
attr *hat*(PERSON) : HAT
attr *contains*(GROUP, PERSON) : `boolean`

The *contains* attribute is non-inert.
Nobody belongs to two groups, and everybody belongs to a group:

dep **ClassMethod**(PERSON, unique)
acc $[t]\neg$override($person$, unique, PERSON) $\wedge$
$group_1$.query_contains($person$) $\wedge$ $group_2$.query_contains($person$) $\rightarrow$
$group_1 = group_2$
dep **ClassMethod**(PERSON, belongs)
acc $[t]\neg$override($person$, belongs, PERSON) $\rightarrow$
$\exists group.[t]\, contains(group, person)$

Finally, we add another rule for modify_group: If *group* moves n of its people to another group $group_2$, then there should be exactly n individual PERSONs that used to belong to *group* but now belong to $group_2$.

acc $[t]\neg$override($group$, modify_group, GROUP) $\wedge$
$[t+1]group$.modify_group($group_2$, $Integer$) $\wedge$ $Integer \geq 0 \wedge$
$group \neq group_2 \rightarrow$
$|\{p \mid p \in$ PERSON $\wedge\, [t]group$.query_contains(p) $\wedge$
$[t+1]group_2$.query_contains(p)$\}| = Integer$

This problem is solved (without exchanging any hats) in 50 seconds.

Four missionaries and four cannibals (elaboration 3)

One cannibal and one missionary are added to the scenario description.

This is a change in the problem instance rather than in the domain, and the instance description is changed accordingly. It is impossible to find a plan for this and the inconsistency is detected in 31 seconds.

Boat can carry three (elaboration 4)

There are four missionaries and four cannibals, but the boat can carry three people.

In the original MCP, the number of people onboard a BOAT was restricted to two. Although it was obvious that it would be useful to be able to model boats of varying capacities, we nonetheless chose to hardcode the capacity in the original boat_limit method in order to test the elaboration tolerance of the model. Thus, we now need to create a subclass that overrides the old constraint. But this time, we will do it the right way:

dom SIZEBOAT *extends* BOAT
attr *capacity*(SIZEBOAT): `Integer`
dep **ClassMethod**(SIZEBOAT, boat_limit)
acc $[t]\neg\mathsf{override}(\mathit{sizeboat}, \mathsf{boat\_limit}, \text{SIZEBOAT}) \rightarrow$
$\quad \mathbf{people\_at}(t, \text{GROUP}, \mathit{value}(t, \mathit{sizeboat}.\mathsf{query\_onboard}())) \leq$
$\quad \mathit{value}(t, \mathit{capacity}(\mathit{sizeboat}))$

A solution is found in 15 seconds (compared to 18 for Lifschitz).

One oar on each bank (elaboration 5)

There is one oar on each bank and there can never be more people than oars in the boat. First we create a class OAR that extends OBJECT. Then we let OARBOAT extend BOAT and add a constraint method ensuring that the sum of people onboard an OARBOAT is less than or equal to the number of OARS:

dep **ClassMethod**(OARBOAT, oar_limit)
dep $[t]\neg\mathsf{override}(\mathit{oarboat}, \mathsf{oar\_limit}, \mathsf{oarboat}) \wedge$
$\quad \mathit{oarboat}.\mathsf{query\_onboard}() = \mathit{place} \rightarrow$
$\quad \mathbf{people\_at}(\text{GROUP}, \mathit{place}) \leq \sum_{\substack{\{g \mid g \in \text{OAR}, \\ [t]g.\mathsf{query\_position}() = \mathit{place}\}}} 1$

We also add a rule that ensures that as many oars as possible are used. If there is someone on the same side as a boat and an oar, the oar will be moved into the boat.

dep **ClassMethod**(OAR, oar_move)
dep $[t]\neg$override(oar, oar_move, OAR) $\wedge$
$boat$.query_position() = $place$ $\wedge$
oar.query_position() = $place$ $\wedge$
$group$.query_position() = $place$ $\wedge$
$group$.query_size() > 0 $\wedge$
$boat$.query_onboard() = $place_2$ $\rightarrow$
$I([t+1]oar$.set_position() $\hat{=}$ $place_2)$

Not everybody can row (elaboration 6 and 7)

Only one missionary and one cannibal can row. This means that we have three new entities to consider: rowing cannibals, rowing missionaries and a boat that only moves if either of those are present.

Three new classes are created. The ROWCAN class extends CANGROUP and ROWMIS extends MISGROUP. None of these add any methods. A new class ROWBOAT is also created that extends BOAT and adds a row_limit constraint:

dep **ClassMethod**(ROWBOAT, row_limit)
dep $[t]\neg$override($rowboat$, row_limit, ROWBOAT) $\wedge$
$rowboat$.query_position() $\neq$ $value(t+1, rowboat$.query_position()) $\wedge$
$rowboat$.query_onboard() = $place$ $\rightarrow$
people_at(ROWCAN, $place$) + **people_at**(ROWMIS, $place$) > 0

This constraint method checks if the boat moves between time-points t and $t+1$ and if so, it checks that the number of rowing cannibals and missionaries is greater than zero.

Elaboration 6: Only one cannibal and one missionary can row. This scenario is solved in 28 seconds compared to Lifschitz' 273 seconds.

Elaboration 7: No missionary can row. The only thing that needs to be changed is the initialization. Contradiction is detected in 3 seconds.

A big cannibal (elaboration 8)

The biggest cannibal cannot fit into the boat with another person.

This can be done in the same way as above. First we construct a new class of cannibals, BIGCANGROUP, that merely extends the CANGROUP class without any extensions and then we extend the BOAT class to a new type of boat, SIZEBOAT that has a size_limit, constraint.

dep **ClassMethod**(SIZEBOAT, size_limit)
dep $[t]\neg$override($sizeboat$, size_limit, SIZEBOAT) $\wedge$
$sizeboat$.query_onboard() = $place$ $\wedge$
$bigcangroup$.query_position() = $place$ $\wedge$
$bigcangroup$.query_size() $> 0 \rightarrow$
people_at(GROUP, $place$) ≤ 1

If there is a BIGCANGROUP group on a boat, and the number of people in that group is more than zero, then the total number of people must be zero or one.

This problem is solved in 614 seconds (compared to Lifschitz' 2149 and 9746 seconds)

Big Cannibal and Small Missionary (elaboration 9)

If the big cannibal is isolated with the smallest missionary, he can eat him.

First we construct the class of small missionaries SMALLMISGROUP, that extends the MISGROUP class without any extensions. Then BIGCANGROUP extends CANGROUP with the additional constraint that the small missionary and big cannibal cannot be allowed to be alone in the same place:

dep **ClassMethod**(BIGCANGROUP, eat_small)
dep $[t]\neg$override($bigcangroup$, eat_small, BIGCANGROUP) $\wedge$
people_at(BIGCANGROUP, $place$) $= 1 \wedge$
people_at(SMALLMISGROUP, $place$) $= 1 \rightarrow$
people_at(GROUP, $place$) > 2

The problem takes 236 seconds to solve (compared to Lifschitz' 22 seconds).

Jesus (elaboration 10)

One of the missionaries is Jesus Christ, who can walk on water. A new group class is created:

dom JESUSGROUP*extends*MISGROUP

The move_persons method from Section 9.4.1 is then overridden with a variation where the condition

$$[t+1]group.\mathsf{query\_pos}().\mathsf{query\_connection}(group_2.\mathsf{query\_pos}())$$

is removed from the precondition, allowing Jesus to move between non-connected places (that is, to cross the river without a boat).

This problem is solved with 6 steps in 4 seconds.

Conversion (elaboration 11)

If three missionaries are isolated with one cannibal, they will convert him. Here we have to take advantage of TAL's possibility to handle true concurrency developed in Chapter 5. Since several different objects can change the value of the size of groups in parallel via modify_group, the constraint method looks like the following

dom CONVMISGROUP *extends* MISGROUP
dep **ClassMethod**(CONVMISGROUP, convert)
dep $[t]\neg\mathsf{override}(convmisgroup, \mathsf{convert}, \text{CONVMISGROUP})\wedge$
$\mathbf{people\_at}(\text{CONVMISGROUP}, place) \geq 3\wedge$
$\mathbf{people\_at}(\text{CANGROUP}, place) = 1 \rightarrow$
$I([t+1]convmisgroup.\mathsf{modify\_group}(convmisgroup) \mathrel{\hat{=}} 1$
$convmisgroup.\mathsf{modify\_group}(cangroup) \mathrel{\hat{=}} -1)$

Note that the actual conflicts that can arise, if for example a cannibal is boarding a boat at the same time as he is converted, are automatically handled by the modify_group method.

This problem is solved in 3 seconds. This cannot be compared to Lifschitz' solution since it does not allow for this kind of concurrency.

Cannibals might steal the boat (elaboration 12)

Whenever a cannibal is alone in the boat, there is a 1/10 probability that he will steal it. Although TAL has no support for probability reasoning, it is possible to determine the probability that any particular boat will be stolen using an attribute *prob_not_stolen* initialized to 1.0. Whenever a cannibal is alone in a boat, the constraint method update_prob multiplies *prob_not_stolen* by 0.9; the value of *boat.prob_not_stolen* at the final time-point of a model is the probability of that particular plan succeeding.

obs $\forall boat.[0]boat.prob\_not\_stolen \hat{=} 1.0$

dep $\mathbf{ClassMethod}(\text{BOAT}, \mathsf{update\_prob})$

dep $[t]\neg\mathsf{override}(boat, \mathsf{update\_prob}, \text{BOAT}) \wedge$
$boat.\mathsf{query\_onboard}() \hat{=} place \wedge$
$\mathbf{people\_at}(t, \text{GROUP}, place) \hat{=} 1 \wedge$
$\mathbf{people\_at}(t, \text{CANGROUP}, place) \hat{=} 1 \rightarrow$
$I([t+1]boat.prob\_not\_stolen \hat{=} 0.9 * value(t, boat.prob\_not\_stolen))$

A plan is found in 6 seconds.

The Bridge (elaboration 13)

There is a bridge from the left bank to the right, which two people can cross at the same time.

The only thing we have to do is to add a BRIDGE class that extends PLACE and adds a constraint bridge_limit that no more than two people can be there at the same time:

dep $\mathbf{ClassMethod}(\text{BRIDGE}, \mathsf{bridge\_limit})$

dep $[t]\neg\mathsf{override}(bridge, \mathsf{bridge\_limit}, \text{BRIDGE}) \rightarrow$
$\mathbf{people\_at}(\text{GROUP}, bridge) \leq 2$

The bridge has to be connected both to the left and right banks. Concurrency falls out naturally, that is, people will walk over the bridge at the same time as others use the boat, without any modifications. This scenario is solved in 22 seconds and requires 5 steps. Again Lifschitz does not allow the use of the bridge and the boat concurrently so the solutions cannot be compared.

The boat leaks (elaboration 14)

The boat leaks and must be bailed concurrently with rowing. A simple way to solve it is by just introducing an attribute *bail* that represent that the boat is being bailed.

dep $\mathbf{ClassMethod}(\text{BAILBOAT}, \mathsf{bailing})$

dep $[t]\neg\mathsf{override}(bailboat, \mathsf{bailing}, \text{BAILBOAT}) \rightarrow$
$group.\mathsf{query\_position}() = bailboat \wedge$
$group.\mathsf{query\_size}())) \geq 1 \rightarrow$
$I([t]bail(group))$

This takes the same time as the basic scenario, 2 seconds.

The boat can be damaged (elaboration 15)

The boat may suffer damage and have to be taken back to the left side for repair. In this elaboration, the boat cannot move between banks instantaneously. We add a new bank **onriver** and a new class SLOWBOAT for boats that spend time on the river before arriving at the destination.

dom SLOWBOAT *extends* BOAT
obj **onriver** : BANK

The original move_boat method is overridden and split into two parts: (1) If the boat is at a BANK and someone is on board, move to **onriver**, and (2) if the boat has been on the river during *crosstime* time-points and there has been no emergency during this interval, move to another bank. The second part takes advantage of TAL's ability to handle delays [18, 49].

First we need to state that if the boat is on a bank and there is someone onboard, then it will move to the river (set_position($slowboat$) $\hat{=}$ **onriver**).

dep **ClassMethod**(SLOWBOAT, move_boat)
dep $[t]\neg$override($slowboat$, move_boat, SLOWBOAT) $\rightarrow$
$slowboat$.query_onboard() $= place_1 \wedge$
$slowboat$.query_position() $= place_2 \wedge$
$place_2 \neq$ **onriver** $\wedge$
people_at(GROUP, $place_1$) $> 0 \rightarrow$
$I([t+1]slowboat$.set_position() $\hat{=}$ **onriver** $\wedge$
$place_1$.remove_connection() $\hat{=} place_2)$

Then we have to describe the normal behavior of a boat on the river:

dep $[t]\neg$override($slowboat$, move_boat, SLOWBOAT) $\wedge$
dep $slowboat$.query_onboard() $= place \wedge$
$slowboat$.query_position() $\neq$ **onriver** $\wedge$
$(t, t + crosstime]slowboat$.query_position()) $\hat{=}$ **onriver**] $\wedge$
$(t, t + crosstime]slowboat$.query_emergency()) $\hat{=} \perp] \rightarrow$
$\exists bank[[t]slowboat$.query_position() $\neq bank \wedge$
$I([t + crosstime]slowboat$.set_position() $\hat{=} bank \wedge$
$place$.add_connection() $\hat{=} bank)]$

This can be interpreted to mean: If at time-point t the boat is on a bank, and the the boat moves out on the river during the *crosstime* following emergency free time-points, then at time-point $t + crosstime$ it will move to a new bank.

Finally we should describe the emergency behavior

dep **ClassMethod**(SLOWBOAT, emergency_behavior)
dep $[t]\neg$override($slowboat$, emergency_behavior, SLOWBOAT)$\wedge$
$slowboat$.query_emergency()$\wedge$
$slowboat$.query_onboard() = $place \rightarrow$
$I([t+3]slowboat$.set_position() $\hat{=}$ **left**$\wedge$
$place$.add_connection() $\hat{=}$ **left**$\wedge$
$slowboat$.set_emergency() $\hat{=} \bot)]$

Naturally each boat has to have an attribute *emergency* and a selector and mutator functions for it. The general constraints have to be modified to take the state of the boat and the duration into consideration. If $crosstime = 3$ and the boat breaks at time 20, this problem is solved in 56 seconds.

The island (elaboration 16)

If an island in the river is added, the problem can be solved with four missionaries and four cannibals. Similar to elaboration 3, we only need to modify the problem instance, the domain is unchanged. We just add another object **island** of type BANK and change the initial size of the left bank group objects.

The complete problem is solved in 5582 seconds (compared to 1894 seconds for Lifschitz' partial solution where only three missionaries and three cannibals end up on the right bank).

Cannibals are not hungry (elaboration 17)

If the strongest cannibal rows fast enough, the cannibals might not get hungry. This is a very vaguely formulated constraint.

The only thing that has changed is the behavior of the cannibals. Therefore we extend that class with a HUNGRYCANGROUP class with a new boolean attribute *hunger*. The method that governs the eating behavior is called eat_constraint, so if we override that with a new method

dep **ClassMethod**(HUNGRYCANGROUP, eat_constraint)
dep $[t]\neg$override($hungrycangroup$, eat_constraint, HUNGRYCANGROUP)$\wedge$
$hungrycangroup$.query_position() = $place\wedge$
$hunger(hungrycangroup)\wedge$
$c\_land =$ **people_at**(CANGROUP, $place$)$\wedge$
$$c\_boat = \sum_{\substack{\{g,b \mid g \in \text{CANGROUP}, b \in \text{BOAT} \wedge \\ [t]b.\text{query\_position}() = place \wedge \\ b.\text{query\_onboard}() = g.\text{query\_position}()\}}} value(t, g.\text{query\_size}()) \wedge$$

$$m\_land = \mathbf{people\_at}(\text{MISGROUP}, place) \wedge$$
$$m\_boat = \sum_{\{g,\ b \ \mid\ g \in \text{MISGROUP},\ b \in \text{BOAT} \wedge [t]b.\mathsf{query\_position}() = place \wedge b.\mathsf{query\_onboard}() = g.\mathsf{query\_position}()\}} value(t, g.\mathsf{query\_size}()) \wedge$$
$$m\_boat + m\_land > 0 \rightarrow$$
$$m\_boat + m\_land \geq c\_land + c\_boat$$

This method is almost exactly the same as the eat_constraint in CANGROUP but with the requirement that the attribute *hunger* has to be be true.

There should also be a new constraint method in HUNGRYCANGROUP saying that they become hungry if the boat changes position and there are no STRONGCANGROUP onboard.

dep $\mathbf{ClassMethod}(\text{HUNGRYCANGROUP}, \mathsf{hunger})$
dep $[t]\neg\mathsf{override}(hungrycangroup, \mathsf{hunger}, \text{HUNGRYCANGROUP}) \wedge$
$boat.\mathsf{query\_onboard}() = place_2 \wedge$
$boat.\mathsf{query\_position}() = place \wedge$
$[t+1]boat.\mathsf{query\_position}() \neq place \wedge$
$\mathbf{people\_at}(\text{STRONGCANGROUP}, place_2) < 1 \rightarrow$
$I([t+1]hunger(hungrycangroup))$

STRONGCANGROUP just extends HUNGRYCANGROUP without any additions.

Missionaries have food (elaboration 18)

The new class FOODCANGROUP extends HUNGRYCANGROUP with a food attribute *hungervalue* and methods that use it:

dep $\mathbf{ClassMethod}(\text{FOODCANGROUP}, \mathsf{feed})$
dep $[t]\neg\mathsf{override}(foodcangroup, \mathsf{feed}, \text{FOODCANGROUP}) \wedge$
$hungervalue(foodcangroup) > 0 \wedge$
$foodcangroup.\mathsf{query\_size}() > 0 \rightarrow$
$I([t+1]hungervalue(foodcangroup) \hat{=}$
$\sum_{\{o \mid o \in \text{OBJECTS}\}} value(t, o.\mathsf{feed}(foodcangroup)) +$
$value(t, hungervalue(foodcangroup)) - 1)$

We let the value of the *hungervalue* attribute in a cannibal be the value in the previous time-point plus the sum of everybody that feeds the cannibals minus one. This means that unless anyone feeds the cannibal its hungervalue will decrease by one each time-point.

The hunger method also has to be overridden with:

dep **ClassMethod**(FOODCANGROUP, hunger)
dep $[t]\neg$override(*foodcangroup*, hunger, FOODCANGROUP) $\wedge$
$hungervalue(foodcangroup) \leq 0 \rightarrow I([t+1]hunger(foodcangroup))$

dep **ClassMethod**(FOODCANGROUP, hunger)
dep $[t]\neg$override(*foodcangroup*, hunger, FOODCANGROUP) $\wedge$
$hungervalue(foodcangroup) > 0 \rightarrow I([t+1]\neg hunger(foodcangroup))$

A public query-function query_hunger is also needed.

dep **ClassMethod**(FOODCANGROUP, query_hunger)
dep $[t]\neg$override(*foodcangroup*, query_hunger, FOODCANGROUP) $\rightarrow$
$I([t]foodcangroup.\text{query\_hunger}() \hat{=} hungervalue(foodcangroup))$

Finally we also want to extend the missionaries to be able to feed the cannibals. Here we cheat and let the cannibals have a global *supply* of food. Let FOODMISGROUP extends MISGROUP and adds the constraint

dep **ClassMethod**(FOODMISGROUP, supplyfood)
dep $[t]\neg$override(*foodmisgroup*, supplyfood, FOODMISGROUP) $\wedge$
$[t]supply > 0 \wedge$
$foodmisgroup.\text{query\_position}() = place \wedge$
$foodmisgroup.\text{query\_size}() > 0 \wedge$
$foodcangroup.\text{query\_position}() = place \wedge$
$foodcangroup.\text{query\_size}() > 0 \wedge$
$value(t, foodcangroup.\text{query\_hunger}()) >$
$value(t+1, foodcangroup.\text{query\_hunger}()) \rightarrow$
$I([t+1]foodmisgroup.\text{feed}(foodcangroup) \hat{=} 2)$

A problem with this is that the hunger is associated to the groups, not to the individuals. This could probably be solved in the same manner as in elaboration 2.

Two sets of people (elaboration 19)

There are two sets of missionaries and cannibals too far apart along the river to interact. A new attribute *connected*(BANK,BANK) keeps track of which banks are connected. Any BOAT moves nondeterministically between all banks. The constraint method move_connected ensures that the origin and destination are connected.

dep **ClassMethod**(BOAT, move_connected)
dep $[t]\neg$override(*boat*, move_connected, BOAT) $\rightarrow$
$boat.\text{query\_pos}().\text{query\_connected}(value(t+1, boat.\text{query\_pos}()))$

This problem is solved in 24 seconds.

9.4.4 Summary

This example shows how the object-oriented approach can be applied to a well known scenario. The inheritance and overriding mechanism provides a simple and familiar way to modify the structure without having to handle all details by hand.

The speed of finding the solutions varies. Generally the time increases with increased branching factors. To find the solution to this kind of problem is a straightforward search problem so loop detection and similar modifications would probably speed up the search immensely. However, the topic of this chapter is not the efficiency of the search, but the modularity and elaboration tolerance of the approach. It should be observed that even without focusing on inference speedup, the system is often faster than the comparable benchmark system developed by Lifschitz et al.

9.5 Object-Oriented Modeling III: The Road Network

The following scenario is inspired by the Traffic World scenario as proposed in the Logic Modelling Workshop [88]. The basic idea is to model cars in a network of roads and crossings. Each car has a top speed and a position. The road segments (the arcs in the graph representing the road-net) also have a speed limit. The actual velocity of the car is the maximum velocity allowed by the following three conditions:

- The speed limit of the road segment where it is driving
- Its own top speed
- The maximum speed where it does not overtake other cars and has a distance of at least δ to the car in front of it.

When a car arrives at a node ("intersection") then it may continue on any arc ("road segment") that connects to that node, except the one it arrived from.

9.5.1 Henschel and Thielscher's Solution

The first model of this scenario was published by Henschel and Thielscher [46] using the Fluent Calculus [100]. Their solution illustrates how the Fluent Calculus can handle complex interactions between ramification, concurrent

actions and continuous change. The only modification to the original Traffic World is that the outgoing cars from a node are located in a queue in a virtual waiting area and portioned out in such a way that the safety distance requirement is not violated on the arcs.

9.5.2 Overview of the Design

TAL currently uses discrete time so our solution naturally has to have the same restriction, but if each time-step is 0.1 second and there is a speed limit of 30 m/s we feel that this restriction is not too severe compared to the other simplifications in the problem formulation itself. As opposed to Henschel and Thielscher's solution we do not need to use virtual waiting areas. If a node is has heavy traffic, cars automatically will queue up on the in-bound arcs of the node.

We have chosen to model this problem using both the object-oriented methodology from Chapter 7 and the control approach from Chapter 8. The classes in the basic scenario are the top class OBJECT, the world classes GRAPH_ELEMENT, ARC, NODE and VEHICLE, and the ego class CONTROLLER.

Object

OBJECT contains no functionality and is only included for completeness.

Graph Element

GRAPH_ELEMENT extends OBJECT and is an intermediate class used for functionality common to both arcs and nodes. The road network is represented with a directed graph. In the current example we name the nodes with one letter and the name of arcs are just concatenations of the start and goal nodes. Since we are dealing with a static road network in this example, we create the connections between arcs and nodes at initialization. The connections of an element are queried with the following method:

- connect(GRAPH_ELEMENT$_1$,GRAPH_ELEMENT$_2$) is true if the arc or node GRAPH_ELEMENT$_1$ is connected to GRAPH_ELEMENT$_2$.

Arc

ARC extends GRAPH_ELEMENT. The distances and speedlimits of the arcs are set in the initialization phase, so the appropriate setters can be skipped.

- distance() returns the length of the arc.
- speedlimit() returns the maximal allowed speed on the arc.

Node

NODE extends GRAPH_ELEMENT. The NODEs represent crossings. To simplify the VEHICLE class and for efficiency reasons the functionality of determining which vehicle is closest to a node is implemented in this class.

- query_closest_leaving() returns the distance to the closest vehicle on an arc from the node. If the node is occupied by a vehicle, that vehicle is returned.
- query_closest_from() is similar to query_closest_leaving(), but does not take vehicles in the node itself into consideration.
- query_distance_leaving() returns the distance from the node to the vehicle returned from query_closest_leaving(). If no vehicle is leaving then the maximal integer is returned.
- query_closest_approaching() returns the closest vehicle on an arc leading into the node.
- query_green_light() returns the vehicle on an approaching arc that has a green light. Here we give green light to the closest approaching vehicle but this function is expected to be overridden with more elaborate decision methods later.

Vehicle

The VEHICLE class extends OBJECT and keeps track of which arc or node it is in and how far on the arc it has traveled. It also takes care of updating the position of the vehicle, taking into consideration its top speed. The attributes are *position*, the distance the vehicle has traveled on an arc, and *topspeed*, the vehicle's maximal velocity.

- set_place(GRAPH_ELEMENT) sets the location of the vehicle to the GRAPH_ELEMENT arc or node.
- query_place() returns the arc or node the vehicle occupies.
- set_position(*Integer*) sets the distance the vehicle has traveled along its arc.
- query_position() returns the distance the vehicle has traveled along its arc.

- move_vehicle(*Integer*) moves the vehicle forward *Integer* units. A vehicle never moves more than its top speed. If it travels past the arc's length, then it is moved into the node.
- set_top_speed(*Integer*) sets the top speed.
- query_closest returns the distance to the vehicle in front of the car. If there is no car on its arc and all arcs leaving the next node then it returns the maximal integer.

Controller

The goal of the CONTROLLER is to govern the behavior of its vehicle. The basic controller only tries to drive 30 units per time point and nondeterministically choose an exit from a node. The attribute of this class is *controls* that refers to the vehicle this controller controls.

Modifications and extensions of this class follows in the next section.

- set_controls(VEHICLE) connects the controller to a VEHICLE
- query_controls() returns the VEHICLE the controller controls.
- proceed() forces the vehicle to proceed to a nondeterministically chosen exit if it is in a node.
- drive() tells the vehicle to try to move 30 units forward.

This controller does not take other vehicles or the nodes query_green_light function into consideration. It just forces the car to move.

9.5.3 Elaborations

Adding Cars, Nodes or Arcs.

Adding more cars, nodes or arcs is trivial. All that needs to be done is to create an object of the correct class and initialize its position and other values.

Drive controller

The controller provided in the basic scenario is far too simplified. The DRIVECONTROLLER illustrates how straightforward it is to use and extend the classes from the basic scenario. DRIVECONTROLLER is an elaboration of CONTROLLER where the drive() method is overridden.

dep **ClassMethod**(DRIVECONTROLLER, drive)
dep $[t]\neg$override(*drivecontroller*, drive, DRIVECONTROLLER) $\wedge$
$drivecontroller$.query_controls() $= vehicle \wedge$
$vehicle$.query_place() $= arc \ \wedge$
$min(value(t, arc.\text{speedlimit}()),$
$value(t, vehicle.\text{query\_closest}()) - 30) = fixpoint \wedge$
arc.query_exit($node$) $\wedge$
$(vehicle.\text{query\_position}() < arc.\text{query\_length}() - 30 \vee$
$node.\text{query\_green\_light}() = arc) \rightarrow$
$I([t]vehicle.\text{move\_vehicle}() \hat{=} fixpoint)$

This prohibits the vehicle from going closer than 30 units to another vehicle. It also makes the vehicle stop before the crossings unless the vehicle has got a green light.

This means that if two vehicles controlled by DRIVECONTROLLERS approach a crossing, one of them will have to stop to let the other pass before continuing.

Deliberate cars

Henschel and Thielscher call the cars controlled by scenario actions *deliberate cars* as opposed to the cars controlled by causal rules. We use the term *action controlled car* for cars that are controlled by actions in the scenario and *object controlled car* for cars that are driven by a controller object. There is no need for us to use different classes to distinguish between these types of cars, the only thing that differs is that there is a controller object associated with the latter.

Timed light node

The class TIMEDLIGHTNODE represents a crossing with a timed traffic-light. All the functionality from the ordinary NODE is kept except the function query_green_light which is overridden with the following implementation:

dep **ClassMethod**(TIMEDLIGHTNODE, query_green_light)
dep $[t]\neg$override(*timedlightnode*, query_green_light, TIMEDLIGHTNODE) $\wedge$
$[t]timedlightnode.\text{query\_green\_light}() = boolean \wedge$
$[t+1]timedlightnode.\text{query\_green\_light}() \neq boolean \rightarrow$
$I([t+30]timedlightnode.\text{query\_green\_light}() \hat{=} boolean)$

It has to be initialized by switching color between time-point 0 and 1.

9.5.4 Summary

This scenario illustrates how an object-oriented approach greatly simplifies a modular construction of larger scenarios. It also shows how an ego can be constructed and then gradually extended with more complex behavior.

Chapter 10

Conclusions and Future Work

The work in this thesis has proposed a number of extensions to the logic TAL. The methodology used has been to make as few modifications as possible to the base logic and to take advantage of the distinction between surface and base languages. Each of the additions has been made incrementally without losing any of the original functionality.

Using this strategy we have studied a number of modeling problems that have been considered to be difficult to solve by the RAC community. One advantage of the methodology used is that the base logic is stable and well understood and that it has already been proven to work for a wide class of problems. It has been thoroughly tested on most benchmark examples proposed by the RAC community, and we have been able to focus on widening the range of applicability of the logic without relinquishing the ability to handle the older set of problems or the functionality it implied. It has also allowed us to develop stable research tools, with a code-base that remains largely unchanged.

One of the red threads in the thesis has been to develop and demonstrate techniques for increasing the modularity and efficiency of the representation. The first step was to incorporate dependency laws, which allow parts of the world model to be represented apart from action descriptions. The actions become more succinct and concise as they only have to describe the preconditions and direct effects, not the possible chains of side effects.

The next step towards modularization was to develop a toolkit of techniques to handle the interaction conflicts that can arise from the use concurrency. This not only allowed us to handle many types of delays, but also provided the basis for developing an object-oriented methodology for narrative construction that allowed us to construct narratives much larger and complex than previously possible in TAL.

Each object has a clearly defined interface to the rest of the model. This technique goes a long way towards guaranteeing modularity and elaboration tolerance. It provides a straightforward basis to exchange, modify or extend objects, as long as they use the correct interface to the rest of the model.

As a result, the TAL family of logic now has a modular and elaboration tolerant solution to the triad of frame, qualification and ramification problems for a large class of applications.

10.1 Future Work

The topic of this thesis has been to extend and develop TAL. The final result has been shown to model problems orders of magnitude more complex than was possible with PMON. The examples in the final chapter of the thesis provide evidence that even much larger problems are within reach.

The TAL family of logics is set up to handle any time structure, but this thesis has focused on discrete time. We have not felt the need to use continuous time in the domains we have been working with since the time-step granularity can be chosen freely. However, investigation into how a continuous time structure can be combined with concurrency and delays is an interesting area of future research.

The surface language of TAL is a macro language which is translated into a second-order logical theory and then reduced to an equivalent first-order logical theory. Information about the domain structure is lost in this translation. With more efficient inference methods it might be possible to take advantage of this structural knowledge and use it to heuristically guide the inference method.

Regarding the work on object-orientation and control, the work presented here is only a first iteration. We have shown how the basic control structures can be modeled, and how controllers can easily be extended incrementally. Using the GOLOG family of languages to solve the same type of problems is an active area of research. It would therefore be very interesting to continue our work with a deeper analysis and comparison between our approach and GOLOG. Another future area of research is to investigate the spectrum between GOLOG-like "programs" that try and reach a goal and the more restricted domain rules used to prune the search space in TLplan-influenced planners [7], such as TALplanner.

All in all there are many different avenues of research available for the continued development of TAL. The requirements of its application domains will continue to govern the direction of the research required in the future.

Appendix A

TAL Without Dependency Laws

In this appendix, we introduce TAL (Temporal Action Logic) without dependency laws and static constraints. It consists of a many-sorted surface language $\mathcal{L}$(ND) (the language for action scenario descriptions), the many-sorted first-order logic $\mathcal{L}$(FL), and translation functions from $\mathcal{L}$(ND) into $\mathcal{L}$(FL). As presented here, it is closer to what historically was called the PMON logic [90, 19, 20]. Dependency laws and static constraints are added in Chapter 4 and formally defined in Appendix B which results in the complete TAL logic.

The first section defines the surface language $\mathcal{L}$(ND), Section 2 defines the base logic $\mathcal{L}$(FL). Section 3 describes the foundational axioms and Section 4 contains the translation function from $\mathcal{L}$(ND) to $\mathcal{L}$(FL) and the minimization policy. Finally Section 5 provides some examples to illustrate the translation process.

A.1 $\mathcal{L}$(ND)

A.1.1 Sorts and Expressions

Definition A.1.1 (Value sorts)
There is a number of sorts $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ for values (including agents and various types of objects); one of these sorts can be the sort $\mathcal{B} = \{\mathsf{true}, \mathsf{false}\}$. For each value sort $\mathcal{V}_i$, there is a corresponding feature sort $\mathcal{F}_i$ for features having values in the value sort. There is also a number of feature symbols f_i, each feature associated with a sort $\mathcal{V}_{i_1} \times \cdots \times \mathcal{V}_{i_n} \rightarrow \mathcal{F}_k$, where $n \geq 0$

is the arity; for each feature symbol, there is an associated value function $value_{f_i}$ of sort $\mathcal{T} \times \mathcal{V}_{i_1} \times \cdots \times \mathcal{V}_{i_n} \rightarrow \mathcal{V}_k$ ($value_{f_i}(\tau, \overline{\omega})$ will often be written $value(\tau, f_i(\overline{\omega}))$). □

Definition A.1.2 (Temporal sort)
There is a temporal sort $\mathcal{T}$ associated with a number of constants 0, 1, 2, 3, ... and $\mathsf{s}_1, \mathsf{t}_1, \ldots$, a function + and three predicates $=$, $<$ and $\leq$. The sort $\mathcal{T}$ is assumed to be interpreted, but can be axiomatized in first-order logic as a subset of Presburger arithmetics [51] (natural numbers with addition). □

Definition A.1.3 (Action sort)
There is a sort for actions $\mathcal{A}$. There is a number of action symbols A_i, each symbol associated with a sort $\mathcal{V}_{i_1} \times \cdots \times \mathcal{V}_{i_n} \rightarrow \mathcal{A}$, where $n \geq 0$ is the arity. □

Definition A.1.4 (elementary time-point expression)
An *elementary time-point expression* (ETE) is a temporal term. □

Definition A.1.5 (elementary value expression)
An *elementary value expression* (EVE) is a value term. □

Definition A.1.6 (elementary feature expression)
An *elementary feature expression* (EFE) is a term of sort $\mathcal{F}_i$ for some i. □

(We will also allow feature symbols to take other EFEs as arguments; this is a shorthand for using the *value* functions and will be removed in the translation to $\mathcal{L}$(FL).)

Definition A.1.7 (Fluents)
Let ϕ be a logic formula. Then, $\mathit{fluents}(\phi)$ is defined as follows:

- If ϕ is of the form $\forall v^i[\psi]$, then $\mathit{fluents}(\phi) = \bigcup_{j=1}^{|\mathcal{V}_i|} \mathit{fluents}(\psi[v^i \mapsto \mathsf{v}^i_j])$.
- If ϕ is of the form $\neg\psi$, then $\mathit{fluents}(\phi) = \mathit{fluents}(\psi)$.
- If ϕ is of the form $\psi \otimes \gamma$, where $\oplus \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$, then $\mathit{fluents}(\phi) = \mathit{fluents}(\psi) \cup \mathit{fluents}(\gamma)$.
- If ϕ is of the form $[\tau]f_k(\omega_1, \ldots, \omega_m) \mathrel{\hat{=}} \Omega$, then $\mathit{fluents}(\phi) = f_k(\omega_1, \ldots, \omega_m)$
- Otherwise $\mathit{fluents}(\phi) = \emptyset$. □

A.1.2 Formulas

Definition A.1.8 (Elementary fluent formula)
An *elementary fluent formula* (EFlF) has the form $f(\omega_1^{i_1}, \ldots, \omega_n^{i_n}) \hat{=} \omega^k$, where f is a feature symbol of sort $\mathcal{V}_{i_1} \times \cdots \times \mathcal{V}_{i_n} \rightarrow \mathcal{F}_k$. If $\mathcal{V}_k = \mathcal{B}$, the formula can also have the form $f(\omega_1^{i_1}, \ldots, \omega_n^{i_n})$, which is a shorthand notation for $f(\omega_1^{i_1}, \ldots, \omega_n^{i_n}) \hat{=} \mathsf{true}$. □

Definition A.1.9 (Fluent formula)
A *fluent formula* (FlF) is an EFlF or a combination of FlFs formed with the standard logical connectives and quantifiers. □

Definition A.1.10 (Fixed fluent formula)
Let τ, τ' be ETEs, ω, ω' be EVEs and α be an FlF. Then $[\tau]\ \alpha$, $\tau = \tau'$, $\tau < \tau'$, $\tau \leq \tau'$ and $\omega = \omega'$ are *fixed formulas* (FF). The formula $[\tau]\ \alpha$ is called a *fixed fluent formula* (FFlF). □

Definition A.1.11 (Logic formula)
A *logic formula* (LF) is an FF or a combination of LFs formed with the standard logical connectives and quantifiers. □

A.1.3 Reassignment

Definition A.1.12 (Elementary reassignment formula)
An *elementary reassignment formula* (ERF) is a formula of the form $[\tau, \tau']\ f(\omega_1^{i_1}, \ldots, \omega_n^{i_n}) := \{\omega_1^k, \ldots, \omega_m^k\}$ or $[\tau, \tau']\ f(\omega_1^{i_1}, \ldots, \omega_n^{i_n}) := \neg\{\omega_1^k, \ldots, \omega_m^k\}$, where f is a feature symbol of sort $[\tau, \tau']\ \mathcal{V}_{i_1} \times \cdots \times \mathcal{V}_{i_n} \rightarrow \mathcal{F}_k$ and all ω_i are value constants or value variables. □

Definition A.1.13 (Restricted reassignment formula)
A *restricted reassignment formula* (RRF) is a conjunction of ERFs with the same time interval. An RRF $\bigwedge_i [\tau, \tau']\ \phi_i$ can be abbreviated $[\tau, \tau']\ \bigwedge_i \phi_i$. □

Definition A.1.14 (Reassignment formula)
A *reassignment formula* (RF) is of the form $(\bigvee_i [\tau, \tau']\ \phi_i \wedge \psi_i)$, where all ϕ_i are RRFs with the same time interval, each EFE occurs in one disjunct iff it occurs in all disjuncts, and each ψ_i is a logic formula representing duration constraints for the time interval. Constraints in ψ may only be placed on the EFEs occurring in ϕ_i, and only during the time interval $(\tau, \tau']$. An RF $(\bigvee_i [\tau, \tau']\ \phi_i) \wedge \psi$ can be abbreviated $[\tau, \tau']\ (\bigvee_i \phi_i) \wedge \psi$. □

Definition A.1.15 (Conditional reassignment formula)
A *conditional reassignment formula* (CRF) is of the form $[\tau]\ \phi(\bar{v}) \rightarrow [\tau, \tau']\ \theta(\bar{v})$, where $\tau \leq \tau'$, $\phi(\bar{v})$ is a fluent formula, and $\theta(\bar{v})$ is a reassignment formula. □

A.1.4 Statements

Definition A.1.16 (Observation statement)
An *observation statement* (labeled obs) is a closed logic formula containing no quantification over time. □

Definition A.1.17 (Action occurrence statement)
An *action occurrence statement* (labeled occ) is an expression of the form $[\tau, \tau']\ A_i$ or $[\tau, \tau']\ A_i(\omega_1^{i_1}, \ldots, \omega_n^{i_n})$, where τ, τ' are ETEs containing no temporal variables, and A_i is an action symbol of arity 0 or of sort $\mathcal{V}_{i_1} \times \cdots \times \mathcal{V}_{i_n} \rightarrow \mathcal{A}$, respectively. □

Definition A.1.18 (Action law schema)
An *action law schema* (labeled acs) is an expression of the form $[t, t']\ A_i \rightsquigarrow \Delta$ or $[t, t']\ A_i(v_1^{i_1}, \ldots, v_n^{i_n}) \rightsquigarrow \Delta$, where t, t' are temporal variables, A_i is an action symbol of arity 0 or of sort $\mathcal{V}_{i_1} \times \cdots \times \mathcal{V}_{i_n} \rightarrow \mathcal{A}$, respectively, and Δ is a conjunction of CRFs where t, t' are the only free variables. □

An action law schema *corresponds* to an action occurrence statement iff they refer to the same action symbol.

Definition A.1.19 (Schedule statement)
A *schedule statement* is a conjunction of CRFs. □

The result of an action occurrence statement $[\tau, \tau']\ A_i(\omega_1^{i_1}, \ldots, \omega_n^{i_n})$ wrt its corresponding action law schema $[t, t']\ A_i(v_1^{i_1}, \ldots, v_n^{i_n}) \rightsquigarrow \Delta$ is $\Delta[t \mapsto \tau][t' \mapsto \tau'][v_1^{i_1} \mapsto \omega_1^{i_1}] \cdots [v_n^{i_n} \mapsto \omega_n^{i_n}]$.

Definition A.1.20 (Action scenario description)
An *action scenario description* (or scenario description) consists of a finite set of:

- Observation statements, prefixed with the label obs,
- Action law schemas, prefixed with the label acs,
- Action occurrence statements, prefixed with the label occ. □

Definition A.1.21 (Expanded action scenario description)
An *expanded action scenario description* consists of a finite set of observation statements and schedule statements. For each action scenario description, the corresponding expanded action scenario description is obtained by replacing each action occurrence statement with its result with respect to its corresponding action law schema and removing all action law schemas. □

A.2 The Base Logic $\mathcal{L}$(FL)

The language $\mathcal{L}$(FL) is a many-sorted first-order language with equality. We use the standard logical connectives $\neg$, $\wedge$, $\rightarrow$, $\leftrightarrow$ and $\vee$, and the standard quantifiers $\forall$ and $\exists$ over temporal, value and fluent variables. We use the sorts from $\mathcal{L}$(ND). For each value sort $\mathcal{V}_i$, we use two predicates $Holds_i$: $\mathcal{T} \times \mathcal{F}_i \times dom(\mathcal{F}_i)$ and $Occlude_i : \mathcal{T} \times \mathcal{F}_i$ (the indices will usually be omitted).

A.3 Foundational Axioms

We will assume that the following axioms are always part of any translation of action scenarios in $\mathcal{L}$(ND) into $\mathcal{L}$(FL). Γ_{UNA} will denote the set of unique names axioms for the action sort, each of the value sorts, and each of the fluent sorts and Γ_{dca} the set of domain closure axioms for the value sorts. In addition, Γ_{val} will denote the set of *value* axioms: For each feature symbol f_i in F of sort $\mathcal{V}_{i_1} \times \ldots, \times \mathcal{V}_{i_n} \rightarrow \mathcal{F}_k$,

$$\forall t. \bigwedge_{\overline{\omega} \in \mathcal{V}_{i_1} \times \cdots \times \mathcal{V}_{i_n},\ \omega^k \in \mathcal{V}_k} \Big(value_{f_i}(t, \overline{\omega}) = \omega^k \ \leftrightarrow \ Holds(t, f_i(\overline{\omega}), \omega^k) \Big)$$

Let $\Gamma_{FA} = \Gamma_{UNA} \cup \Gamma_{dca} \cup \Gamma_{val}$.

A.4 Translation Functions

EVEs ω in $\mathcal{L}$(ND) are translated using the $Tran_{EVE}$ function:
If $\omega = value(\tau, f_k(\omega_1, \ldots, \omega_n))$, then $Tran_{EVE}(\omega) = value_{f_k}(\tau, Tran_{EVE}(\tau, \omega_1), \ldots, Tran_{EVE}(\tau, \omega_n))$; otherwise $Tran_{EVE}(\omega) = \omega$.

Value expressions ω in $\mathcal{L}$(ND) are translated using the $Tran_{VE}$ function: If $\omega = f_k(\omega_1, \ldots, \omega_n)$, then $Tran_{VE}(\tau, \omega) = value_{f_k}(\tau, Tran_{EVE}(\tau, \omega_1), \ldots, Tran_{EVE}(\tau, \omega_n))$; otherwise $Tran_{VE}(\tau, \omega) = Tran_{EVE}(\omega)$.

EFEs f in $\mathcal{L}$(ND) are translated using the $Tran_{EFE}$ function: If $f = f_k(\omega_1, \ldots, \omega_n)$, then $Tran_{EFE}(\tau, \omega) = f_k(Tran_{EVE}(\tau, \omega_1), \ldots, Tran_{EVE}(\tau, \omega_n))$.

Logic formulas in $\mathcal{L}$(ND) are translated to $\mathcal{L}$(FL) by applying the function $Tran_{LF}$, defined by the following rewrite rules:

$$\begin{aligned}
[\tau]\neg\alpha &\stackrel{\text{def}}{=} \neg[\tau]\ \alpha \\
[\tau]\alpha \otimes \beta &\stackrel{\text{def}}{=} [\tau]\ \alpha \otimes [\tau]\ \beta \\
[\tau]\forall v[\alpha] &\stackrel{\text{def}}{=} \forall v[[\tau]\ \alpha] \\
[\tau]\exists v[\alpha] &\stackrel{\text{def}}{=} \exists v[[\tau]\ \alpha] \\
\omega = \omega' &\stackrel{\text{def}}{=} Tran_{EVE}(\omega) = Tran_{EVE}(\omega') \\
[\tau]f \hat{=} \{\omega_1, \ldots, \omega_m\} &\stackrel{\text{def}}{=} \bigvee_{j=1}^{m} Holds(\tau, Tran_{EFE}(f), Tran_{EVE}(\omega_j))
\end{aligned}$$

Reassignment formulas in $\mathcal{L}$(ND) are translated to $\mathcal{L}$(FL) by applying the following rewrite rules, where t is a fresh temporal variable:

$$[\tau, \tau']\ f := \Omega \stackrel{\text{def}}{=} Tran_{LF}([\tau']\ f \hat{=} \Omega) \wedge \forall t.\tau < t \leq \tau' \rightarrow Occlude(t, Tran_{EFE}(f))$$

$$[\tau, \tau']\ f := \neg\Omega \stackrel{\text{def}}{=} \neg Tran_{LF}([\tau']\ f \hat{=} \Omega) \wedge \forall t.\tau < t \leq \tau' \rightarrow Occlude(t, Tran_{EFE}(f))$$

The *Nochange axiom* Γ_{NCG} is used as a filter, to remove models with change that is not justified. Γ_{NCG} is the union of all Γ_{NCG_i} such that for each $\mathcal{F}_i$:

$$_{NCG_i} = \forall t, f^i, v^i(\neg Occlude(t+1, f^i) \rightarrow Holds(t, f^i, v^i) \leftrightarrow Holds(t+1, f^i, v^i))$$

Let Υ be an action scenario in $\mathcal{L}$(ND). Let Γ_{OBS} and Γ_{SCD} denote the translations into $\mathcal{L}$(FL) of the observations and the expanded action laws from Υ, respectively. In addition let Γ_{fnd} denote the foundational axioms which include the unique names axioms, Γ_{UNA} and the axioms for the time structure. Let $Trans(\Upsilon)$ be the translation of Υ into $\mathcal{L}$(FL). Then the formula α is entailed by $Trans(\Upsilon)$ iff

$$\Gamma_{fnd} \wedge \Gamma_{NCG} \wedge \Gamma_{OBS} \wedge Circ_{SO}(\Gamma_{SCD}; Occlude) \models \alpha$$

Since *Occlude* only occurs positively in Γ_{SCD}, we are guaranteed a reduction to first-order logic. In practice we can use predicate completion instead of circumscription.

A.5 Examples

Example A.5.1 (Yale Shooting Scenario, Hanks and McDermott [44]) This example shows the translation from surface language to $\mathcal{L}(\mathrm{FL})$ of the Yale Shooting Problem.

obs_1 $[0]\mathsf{alive} \wedge \neg\mathsf{loaded}$

occ_1 $[2,4]$ Load

occ_2 $[5,6]$ Fire

acs_1 $[t_1,t_2]$ Load $\rightsquigarrow$ $[t_1,t_2]$ loaded $:=$ true

acs_2 $[t_1,t_2]$ Fire $\rightsquigarrow$ ($[t_1]$ loaded $\rightarrow$ $[t_1,t_2]$ (loaded $:=$ false $\wedge$ alive $:=$ false)).

After expansion we get:

obs_1 $[0]\mathsf{alive} \wedge \neg\mathsf{loaded}$

acs_1 $[2,4]\mathsf{loaded} := \mathsf{true}$

acs_2 $[5]$ loaded $\rightarrow$ $[5,6]$ (loaded $:=$ false $\wedge$ alive $:=$ false).

The corresponding set of labeled wffs in $\mathcal{L}(\mathrm{FL})$ is

obs_1 $\mathit{Holds}(0, \mathsf{alive}, \mathsf{true}) \wedge \neg \mathit{Holds}(0, \mathsf{loaded}, \mathsf{true})$

acs_1 $\mathit{Holds}(4, \mathsf{loaded}, \mathsf{true}) \wedge (\forall t.2 < t \leq 4 \rightarrow \mathit{Occlude}(t, \mathsf{loaded}))$

acs_2 $\mathit{Holds}(5, \mathsf{loaded}, \mathsf{true}) \rightarrow$
$[(\mathit{Holds}(6, \mathsf{loaded}, \mathsf{false}) \wedge \mathit{Holds}(6, \mathsf{alive}, \mathsf{false})) \wedge$
$\mathit{Occlude}(6, \mathsf{loaded}) \wedge \mathit{Occlude}(6, \mathsf{alive})]$.

Circumscription of Γ_{SCD} with *Occlude* minimized results in the following:

$$\forall t, f.\mathit{Occlude}(t, f) \leftrightarrow$$
$$[(2 < t \leq 4 \wedge f = \mathsf{loaded}) \vee$$
$$(\mathit{Holds}(5, \mathsf{loaded}, \mathsf{true}) \wedge t = 5 \wedge (f = \mathsf{alive} \vee f = \mathsf{loaded}))] \qquad \square$$

The most important observations about TAL is that it permits scenarios with nondeterministic actions, actions with duration, partial specification at any state in the scenario, context dependency, and incomplete specification on the timing and order of actions.

Appendix B

Modifications of TAL for Dependency Laws

This appendix contains the definitions of dependency laws and the modified minimization policy.

B.1 Dependency Laws

Definition B.1.1
A *dependency law* has the form

$$\forall t_1, \overline{v}[\gamma \rightarrow ([\tau]\alpha \gg [\tau']\beta)], \tag{B.1}$$

where

- τ and τ' are ETEs,
- τ is a function of t_1 and is greater than or equal to t_1,
- τ' is a function of τ and is greater than or equal to τ,
- γ, the *precondition*, is a LF that does not contain conditions on fluents outside the time interval $[\tau, \tau']$,
- $[\tau]\alpha$, the *trigger*, is a FFlF with no occurrences of *value*, and
- $[\tau']\beta$, the *postcondition*, is a FFlF where all EFlFs are non-nested. □

If $\tau < \tau'$, we call the dependency law a delayed dependency law.

Dependency laws will be labeled with the prefix *dep* in action scenario descriptions in $\mathcal{L}$(ND).

The formula $\forall t_1, \overline{v}([\tau]\alpha \gg [\tau']\beta)$ may be used as a dependency law as an abbreviation of $\forall t_1, \overline{v}[\mathsf{true} \rightarrow ([\tau]\alpha \gg [\tau']\beta)]$. If $\tau = \tau'$ we may also write $[\tau](\alpha \gg \beta)$.

The following macro-translation definition provides the proper translation for the dependency law from $\mathcal{L}(ND)$ into the logic $\mathcal{L}(FL)$.

Definition B.1.2
Let

$$\Phi = \forall t_1, \overline{v}[\gamma \rightarrow ([\tau]\alpha \quad \gg \quad [\tau']\beta)]$$

be a dependency law. Then,

$$\begin{aligned} Tran(\Phi) = \quad & \forall t_1, \overline{v}\ [Tran_{LF}(\gamma \wedge [\tau]\alpha \rightarrow [\tau']\beta)] \wedge & \text{(B.2)} \\ & \forall t_1, z, \overline{v}[(Tran_{LF}(0 < t_1 \wedge \gamma \wedge \forall t'(t' + 1 = \tau \rightarrow [t']\neg\alpha) \wedge \\ & \qquad [\tau]\alpha) \wedge \bigvee_{f \in fluents(\beta)} z = f) \rightarrow \\ & \qquad Occlude(\tau', z)]. & \text{(B.3)} \end{aligned}$$

□

B.2 Minimization Policy

Static constraints are translated using $Tran_{LF}$. Let Γ_{ACC} and Γ_{DEP} denote the translations into $\mathcal{L}$(FL) of the static constraints and the dependency laws in an action scenario.

Let Υ be and action scenario in $\mathcal{L}$(ND), and $Trans(\Upsilon)$ be its translation into $\mathcal{L}$(FL). Then the formula α is entailed by $Trans(\Upsilon)$ iff

$$\Gamma_{fnd} \wedge \Gamma_{NCG} \wedge \Gamma_{OBS} \wedge \Gamma_{ACC} \wedge Circ_{SO}(\Gamma_{SCD} \wedge \Gamma_{DEP}; Occlude) \models \alpha \qquad \text{(B.4)}$$

Note that the the *Occlude* predicate only occurs positively in a translation of an action scenario. Consequently, we can show that the circumscription above is reducible to a logically equivalent first-order formula.

Appendix C

Formal Specification of TAL-C

In TAL-C we have chosen to change some of the notation from earlier versions of TAL, in order to make the language more consistent. Also, more importantly, the changes allow us to generalize the form of the dependency laws. Section C.1 contains the definition of the surface language used in TAL-C. The base language $\mathcal{L}$(FL) and a translation from the surface language to $\mathcal{L}$(FL) is provided in Section C.2. Finally Section C.3 contains an example of the translation process.

C.1 The Language $\mathcal{L}$(ND) for TAL-C

Definition C.1.1 (Value sorts)
There are a number of sorts for values $\mathcal{V}_i$ (including agents and various types of objects). One of these sorts is the boolean sort $\mathcal{B}$ with the constants $\{\mathsf{true}, \mathsf{false}\}$. Further, there are a number of sorts for features $\mathcal{F}_i$, each one associated with a value domain $dom(\mathcal{F}_i) = \mathcal{V}_j$ for some j. □

Definition C.1.2 (Action sort)
There is a sort for actions $\mathcal{A}$. □

Definition C.1.3 (Temporal sort)
Finally, there is a temporal sort $\mathcal{T}$ associated with a number of constants $0, 1, 2, 3, \ldots$ and $\mathsf{s}_1, \mathsf{t}_1, \ldots$ and a function $+$ and three predicates $=$, $<$ and $\leq$. $\mathcal{T}$ is assumed to be an interpreted sort, but can be axiomatized in first-

order logic as a subset of Presburger arithmetics [51] (natural numbers with addition). □

Definition C.1.4 (Time-point expression and temporal formula)
A *time-point expression* is a temporal term. If τ, τ' are time-point expressions, then $\tau = \tau'$, $\tau < \tau'$ and $\tau \leq \tau'$ are *temporal formulas.* □

Definition C.1.5 (Value formula)
A *value formula* is of the form $\omega = \omega'$ where ω and ω' are value terms. □

Definition C.1.6 (Elementary fluent formula)
An *elementary fluent formula* has the form $f(\overline{\omega})\hat{=}v$, where f is a feature name of sort $\mathcal{V}_1 \times \ldots \times \mathcal{V}_n \rightarrow \mathcal{F}_i$ and $\overline{\omega}$ is a vector of value terms such that each term ω_i is of sort $\mathcal{V}_i$ and v is a term of sort $dom(\mathcal{F}_i)$. It can also have the form $f(\overline{\omega})$ if $dom(\mathcal{F}_i) = \mathcal{B}$, or $f\hat{=}v$ if f is a feature variable. □

Definition C.1.7 (Fluent formula)
A *fluent formula* is an elementary fluent formula or a combination of fluent formulas formed with the standard logical connectives and quantifiers. □

Definition C.1.8 (Fixed fluent formula and becomes formula)
Let τ, τ' be time-point expressions and α a fluent formula. Then $[\tau, \tau']\alpha$ and $[\tau]\alpha$ are *fixed fluent formulas*, and $C_T([\tau]\alpha)$ is a *becomes formula.* □

Definition C.1.9 (Reassignment, interval and occlusion formula)
Let τ, τ' be time-point expressions and α a fluent formula. $R((\tau, \tau']\alpha)$ and $R([\tau]\alpha)$ are *reassignment formulas*, $I((\tau, \tau']\alpha)$ and $I([\tau]\alpha)$ are *interval formulas*, and $X((\tau, \tau']\alpha)$ and $X([\tau]\alpha)$ are *occlusion formulas.* □

Those formulas with one time-point expression are called *instantaneous*, and those with two are called *time spanning.*

Definition C.1.10 (Occurrence formula)
An *occurrence formula* has the form $[\tau, \tau']\Phi(\overline{\omega})$, where τ and τ' are elementary time-point expressions, Φ is an action name of sort $\mathcal{V}_1 \times \ldots \times \mathcal{V}_n \rightarrow \mathcal{A}$ and the value terms in $\overline{\omega}$ are of matching sorts. □

Definition C.1.11 (Static formula)
A logical combination of quantifiers, temporal and value formulas, fixed fluent formulas and/or becomes formulas is called a *static formula.* □

Definition C.1.12 (Change formula and balanced change formula)
A *change formula* is a formula that has the form, or is rewritable to the form, $\mathcal{Q}(\alpha_1 \vee \cdots \vee \alpha_n)$ where $\mathcal{Q}$ is a set of quantifiers, and each α_i is a conjunction of static, occlusion, reassignment and interval formulas. The change formula is called *balanced* iff (a) whenever a feature $f(\overline{\omega})$ appears inside a reassignment, interval or occlusion formula in one of the α_i conjuncts, then it must also appear in all other α_i's inside a reassignment, interval or occlusion formula with exactly the same temporal argument; and (b) any existentially quantified variable v in the formula, whenever appearing inside a reassignment, interval or occlusion formula, only does so in the position $f(\overline{\omega}) \hat{=} v$. □

Definition C.1.13 (Application formula)
An *application formula* is any of the following:

1. A balanced change formula.
2. $\Lambda \rightarrow \Delta$, where Λ is a static formula and Δ is a balanced change formula.
3. A conjunction of application formulas.
4. $\forall t[\Phi]$, where Φ is an application formula.
5. $\forall v[\Phi]$, where Φ is an application formula. □

An action law (labeled acs) has the form $[t, t']\Phi(\overline{a}) \rightarrow \Psi$ where $[t, t']\Phi(\overline{a})$ is an occurrence formula, Ψ is an application formula in which the variables in $\overline{a}$ may occur free. A dependency law (labeled dep) is an application formula. An observation (labeled obs) is a static formula. An occurrence (labeled occ) is an occurrence formula $[\tau, \tau']\Phi(\overline{\omega})$ where τ, τ', $\overline{\omega}$ all are constants. A domain formula (labeled dom) is a universally quantified conjunction of *Per* and *Dur* statements and other domain-specific statements of choice, universally quantified over time.

Finally, a *scenario* is a tuple $\langle dom, law, dep, obs, occ \rangle$ where each element is a set of statements with the corresponding labeling.

C.2 Translation from $\mathcal{L}$(ND) to $\mathcal{L}$(FL)

$\mathcal{L}$(FL) is a first-order language consisting of the predicates $Holds_i : \mathcal{T} \times \mathcal{F}_i \times Dom(\mathcal{F}_i)$, $Occlude_i : \mathcal{T} \times \mathcal{F}_i$ (normally, the index i is omitted) and $Occurs : \mathcal{T} \times \mathcal{T} \times \mathcal{A}$, and all predicates relating to the value domains and temporal domain from $\mathcal{L}$(ND). There is an isomorphism of sorts and names between $\mathcal{L}$(ND) and $\mathcal{L}$(FL).

Definition C.2.1
Tran is called the *expansion transformation*, and is defined as follows (the obvious parts have been left out). All variables occurring only on the right-hand side are assumed to be fresh variables.

$$
\begin{aligned}
Tran([\tau]f(\overline{\omega})) &= Holds(\tau, f(\overline{\omega}), \mathsf{true}) && \text{(C.1)}\\
Tran([\tau]f(\overline{\omega}) \hat{=} v) &= Holds(\tau, f(\overline{\omega}), v) && \text{(C.2)}\\
Tran([\tau]\neg\alpha) &= \neg\, Tran([\tau]\alpha) && \text{(C.3)}\\
Tran([\tau]\alpha \mathcal{C} \beta) &= Tran([\tau]\alpha)\ \mathcal{C}\ Tran([\tau]\beta) && \text{(C.4)}\\
&\quad \text{where } \mathcal{C} \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}.\\
Tran([\tau]\mathcal{Q}v[\alpha]) &= \mathcal{Q}v[Tran([\tau]\alpha)] \text{ where } \mathcal{Q} \in \{\forall, \exists\}. && \text{(C.5)}\\
Tran([\tau, \tau']\alpha) &= \forall t'[\tau \leq t' \leq \tau' \rightarrow Tran([t']\alpha)] && \text{(C.6)}\\
Tran(C_T([\tau]\alpha)) &= \forall t'[\tau = t' + 1 \rightarrow Tran([t']\neg\alpha)] \wedge && \text{(C.7)}\\
&\quad Tran([\tau]\alpha)\\
Tran(X([\tau]f(\overline{\omega}))) &= Occlude(\tau, f(\overline{\omega})) && \text{(C.8)}\\
Tran(X([\tau]f(\overline{\omega}) \hat{=} v)) &= Occlude(\tau, f(\overline{\omega})) && \text{(C.9)}\\
Tran(X([\tau]\neg\alpha)) &= Tran(X([\tau]\alpha)) && \text{(C.10)}\\
Tran(X([\tau]\alpha \mathcal{C} \beta)) &= Tran(X([\tau]\alpha)) \wedge Tran(X([\tau]\beta)) && \text{(C.11)}\\
&\quad \text{where } \mathcal{C} \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}.\\
Tran(X([\tau]\mathcal{Q}v[\alpha])) &= \forall v[Tran(X([\tau]\alpha))] \text{ where } \mathcal{Q} \in \{\forall, \exists\}. && \text{(C.12)}\\
Tran(X((\tau, \tau']\alpha)) &= \forall t'[\tau < t' \leq \tau' \rightarrow Tran(X([t']\alpha))] && \text{(C.13)}\\
Tran(R((\tau, \tau']\alpha)) &= Tran(X((s, t], \alpha)) \wedge Tran([\tau]\alpha) && \text{(C.14)}\\
Tran(R([\tau]\alpha)) &= Tran(X([\tau], \alpha)) \wedge Tran([\tau]\alpha) && \text{(C.15)}\\
Tran(I((\tau, \tau']\alpha)) &= Tran(X((\tau, \tau']\alpha)) \wedge Tran([\tau + 1, \tau']\alpha) && \text{(C.16)}\\
Tran(I([\tau]\alpha)) &= Tran(X([\tau], \alpha)) \wedge Tran([\tau]\alpha) && \text{(C.17)}\\
Tran([\tau, \tau']\Phi(\overline{\omega})) &= Occurs(\tau, \tau', \Phi(\overline{\omega})) && \text{(C.18)}
\end{aligned}
$$

□

Notice the translation of the X operator, in particular lines (C.10–C.12), which will always occlude all features referenced inside a reassignment or interval formula.

The second-order circumscription of a number of predicates $\overline{P} = P_1, \ldots, P_n$ in the theory $\Gamma(\overline{P})$ is denoted $Circ_{SO}(\Gamma(\overline{P}); \overline{P})$ [60]. Intuitively, the formula $Circ_{SO}(\Gamma(\overline{P}); \overline{P})$ represents a (second-order) theory containing $\Gamma(\overline{P})$ and where the extensions of the predicates $\overline{P}$ are minimal.

Definition C.2.2
Transformation of scenarios from $\mathcal{L}$(ND) to $\mathcal{L}$(FL):

1. Let *dom*, *acs*, *dep*, *obs* and *occ* be the sets of statements with labels dom, acs, dep, obs and occ respectively, completed with universal quantification for variables occurring freely.

2. Let $\Gamma_{dom} = Tran(dom)$, $\Gamma_{acs} = Tran(acs)$, $\Gamma_{dep} = Tran(dep)$, $\Gamma_{obs} = Tran(obs)$ and $\Gamma_{occ} = Tran(occ)$.

3. Let Γ be $Circ_{SO}((\Gamma_{acs} \cup \Gamma_{dep})(\overline{Occlude}); \overline{Occlude}) \cup \Gamma_{dom} \cup Circ_{SO}(\Gamma_{occ}(Occurs); Occurs) \cup \Gamma_{obs} \cup \Gamma_{fl} \cup \Gamma_{fnd}$. Γ is the theory that is used for proofs in TAL-C. □

The set Γ_{fl} contains the equivalent of the TAL nochange axiom enhanced to deal with durational features, plus two more axioms relating to *Per* and *Dur*.

$$
\begin{aligned}
\Gamma_{fl} \;=\; & \bigcup_i \{ \qquad\qquad (C.19)\\
& \forall f_i, t, v_i \,[\, Dur_i(f_i, t, v_i) \rightarrow \\
& \qquad (\neg Occlude_i(t, f_i) \rightarrow (Holds_i(t, f_i, v_i)))],\\
& \forall f_i, t, v_i [\, Per_i(f_i, t) \rightarrow (\neg Occlude_i(t+1, f_i) \rightarrow \\
& \qquad (Holds_i(t, f_i, v_i) \leftrightarrow Holds_i(t+1, f_i, v_i)))],\\
& \forall f_i, t, v_i, v_i' \; Dur_i(f_i, t, v_i) \wedge Dur_i(f_i, t, v_i') \rightarrow v_i = v_i',\\
& \forall f_i, t [Per_i(f_i, t) \oplus \exists v_i \, Dur_i(f_i, t, v_i)] \;\}
\end{aligned}
$$

Finally, the set Γ_{fnd} consists of foundational axioms for unique names for actions, features and values, and constraints that a feature has exactly one value at each time-point.

An important property of the circumscribed theory Γ is that although it is a second-order theory due to the second-order nature of circumscription, it can be reduced to an equivalent first-order theory, and in a very convenient form. The following is a principal account for this reduction; for details, the proofs in [21] are directly applicable. Due to the definition of action laws and dependency laws occlusion can only occur on the right-hand side Δ of an implication $\Gamma \rightarrow \Delta$. Furthermore, due the restrictions on balanced change formulas and the definition of the *Tran* function, occlusion only occurs positive in Δ, and if it occurs in a disjunction inside Δ, then it occurs identically on both sides. Therefore, the $Occlude_i$ parts, using the law of distributivity, can be separated from the rest of Γ, giving $\Lambda \rightarrow (\bigwedge_i \Delta_i^{occ}) \wedge \Delta^h$, and from there to $\Lambda \rightarrow \Delta^h \wedge (\bigwedge_i \Lambda \rightarrow \Delta_i^{occ})$. Thus, it

can be shown that each expanded action law or dependency law is equivalent to a formula $\Lambda \rightarrow \Delta^h \wedge (\bigwedge_i \forall t, f_i[\Lambda' \rightarrow Occlude_i(t, f_i)])$.

Now, there are two useful theorems by Lifschitz [63], the first stating that if B does not contain P, then $Circ_{SO}(\Gamma(P) \wedge B; P) \leftrightarrow Circ_{SO}(\Gamma(P); P) \wedge B$, and the second stating that if $F(\overline{x})$ does not contain P, then $Circ_{SO}(\forall \overline{x} F(\overline{x}) \rightarrow P(\overline{x}); P) \leftrightarrow (\forall \overline{x} F(\overline{x}) \leftrightarrow P(\overline{x}))$. From these theorems and the equivalent form above it follows that $Circ_{SO}((\Gamma_{law} \cup \Gamma_{dep})(\overline{Occlude}); \overline{Occlude}) = (\bigwedge_k \Lambda_k \rightarrow \Delta_k^h) \wedge (\bigwedge_i \forall t, f_i[(\bigvee_k \Lambda'_{ik}) \leftrightarrow Occlude_i(t, f_i)])$, which is first-order.

C.3 Example

In order to illustrate the translation function from the previous section, an example scenario in TAL-C is transformed from $\mathcal{L}$(ND) to $\mathcal{L}$(FL).

Example C.3.1
This is a complete translation of a scenario with two actions to $\mathcal{L}$(FL) and minimization of the *Occurs* and *Occlude* predicates. The first action LightFire(person, thing) expresses the fact that person is trying to light thing. The second action PourWater(person, thing) means that person is pouring water on thing. The first and second of the dependency laws describe the behavior of the burning feature (in addition to the first domain statement which declares burning to be persistent), whereas the third describes the behavior of the dry feature. The associated influences are fire-infl and wet-infl.

dom_1	$\forall t(Per(\mathsf{dry}(x), t) \wedge Per(\mathsf{burning}(x), t) \wedge$
	$Dur(\mathsf{fire\text{-}\ infl}(a, x), \mathsf{false}, t) \wedge Dur(\mathsf{wet\text{-}infl}(a, x), \mathsf{false}, t))$
acs_1	$[s, t]\mathsf{LightFire}(a, x) \rightarrow I((s, t]\mathsf{fire\text{-}infl}(a, x))$
acs_2	$[s, t]\mathsf{PourWater}(a, x) \rightarrow I((s, t]\mathsf{wet\text{-}infl}(a, x))$
dep_1	$[s, s+3](\mathsf{fire\text{-}infl}(a, x) \wedge \mathsf{dry}(x) \wedge \mathsf{wood}(x)) \rightarrow$
	$R([s+3]\mathsf{burning}(x))$
dep_2	$[s]\neg\mathsf{dry}(x) \rightarrow R([s]\neg\mathsf{burning}(x))$
dep_3	$[s]\mathsf{wet\text{-}infl}(a, x) \rightarrow R([s]\neg\mathsf{dry}(x))$
obs_1	$[0]\mathsf{dry}(\mathsf{wood1}) \wedge \neg\mathsf{burning}(\mathsf{wood1}) \wedge \mathsf{wood}(\mathsf{wood1})$
occ_1	$[2, 7]\mathsf{LightFire}(\mathsf{bill}, \mathsf{wood1})$
occ_2	$[5, 6]\mathsf{PourWater}(\mathsf{bob}, \mathsf{wood1})$

Translation to $\mathcal{L}$(ND) yields

$$
\begin{aligned}
\Gamma_{law} \;=\; \{ & \\
& \forall s,t,a,x.Occurs(s,t,\mathsf{LightFire}(a,x)) \rightarrow \\
& \qquad \forall t'[s < t' \leq t \rightarrow Holds(t',\mathsf{fire\text{-}infl}(a,x),\mathsf{true})] \wedge \\
& \qquad \forall t'[s < t' \leq t \rightarrow Occlude(t',\mathsf{fire\text{-}infl}(a,x))], \\
& \forall s,t,a,x.Occurs(s,t,\mathsf{PourWater}(a,x)) \rightarrow \\
& \qquad \forall t'[s < t' \leq t \rightarrow Holds(t',\mathsf{wet\text{-}infl}(a,x),\mathsf{true})] \wedge \\
& \qquad \forall t'[s < t' \leq t \rightarrow Occlude(t',\mathsf{wet\text{-}infl}(a,x))], \\
& \forall s,a,x(\forall t'[s \leq t' \leq s+3 \rightarrow (Holds(t',\mathsf{fire\text{-}infl}(a,x),\mathsf{true}) \wedge \\
& \qquad Holds(t',\mathsf{dry}(x),\mathsf{true}) \wedge Holds(t',\mathsf{wood}(x),\mathsf{true}))] \rightarrow \\
& \qquad\quad Occlude(s+3,\mathsf{burning}(x)) \wedge Holds(s+3,\mathsf{burning}(x),\mathsf{true})), \\
& \forall s,a,x.\neg Holds(s,\mathsf{dry}(x),\mathsf{true}) \rightarrow \\
& \qquad (\, Occlude(s,\mathsf{burning}(x)) \wedge \neg Holds(s,\mathsf{burning}(x),\mathsf{true})), \\
& \forall s,a,x.Holds(s,\mathsf{wet\text{-}infl}(a,x),\mathsf{true}) \rightarrow \\
& \qquad (\, Occlude(s,\mathsf{dry}(x)) \wedge \neg Holds(s,\mathsf{dry}(x),\mathsf{true}))\}
\end{aligned}
$$

$$
\begin{aligned}
\Gamma_{occ} \;=\; & \{Occurs(2,7,\mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})), \\
& \qquad Occurs(5,6,\mathsf{PourWater}(\mathsf{bob},\mathsf{wood1}))\}
\end{aligned}
$$

$$
\begin{aligned}
\Gamma_{obs} \;=\; & \{Holds(0,\mathsf{dry}(\mathsf{wood1}),\mathsf{true}) \wedge \neg Holds(0,\mathsf{burning}(\mathsf{wood1}),\mathsf{true}) \wedge \\
& \quad Holds(0,\mathsf{wood}(\mathsf{wood1}),\mathsf{true})\}
\end{aligned}
$$

Circumscribing $Occurs$ in Γ_{occ} and $Occlude$ in $\Gamma_{law} \cup \Gamma_{dep}$ yields the following exact descriptions of the two predicates, which together with the original theory and the additional components constitute Γ. Notice that the $Occlude$ part specifies exactly the exceptions to the default rules for persistent and durational features, as expressed in the two first axioms in Γ_{fl}.

$$
\begin{aligned}
& \forall s,t,e\ [Occurs(s,t,e) \leftrightarrow \\
& \qquad ((s = 2 \wedge t = 7 \wedge e = \mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})) \vee \\
& \qquad (s = 5 \wedge t = 6 \wedge e = \mathsf{PourWater}(\mathsf{bob},\mathsf{wood1})))]
\end{aligned}
$$

$$\forall t', f\ (Occlude(t', f) \leftrightarrow \exists s, t, a, x[$$
$$((s < t' \leq t \wedge f = \text{fire-infl}(a, x) \wedge Occurs(s, t, \text{LightFire}(a, x))) \vee$$
$$(s < t' \leq t \wedge f = \text{wet-infl}(a, x) \wedge Occurs(s, t, \text{PourWater}(a, x))) \vee$$
$$(t' = s + 3 \wedge f = \text{burning}(x) \wedge \forall s'[s \leq s' \leq s + 3 \rightarrow$$
$$Holds(t', \text{fire-infl}(a, x), \text{true}) \wedge Holds(t', \text{dry}(x), \text{true})]) \vee$$
$$(f = \text{burning}(x) \wedge \neg Holds(t', \text{dry}(x), \text{true})) \vee$$
$$(f = \text{dry}(x) \wedge Holds(t', \text{wet-infl}(a, x), \text{true})))])$$

Below is the complete theory except Γ_{fl} and Γ_{fnd}:

$$\forall t(Per(\text{dry}(x), t) \wedge Per(\text{burning}(x), t) \wedge$$
$$Dur(\text{fire-infl}(a, x), t, \text{false}) \wedge Dur(\text{wet-infl}(a, x), t, \text{false})) \wedge \quad \text{(C.20)}$$
$$\forall s, t, a, x.Occurs(s, t, \text{LightFire}(a, x)) \rightarrow$$
$$\forall t'[s < t' \leq t \rightarrow Holds(t', \text{fire-infl}(a, x), \text{true})] \wedge \quad \text{(C.21)}$$
$$\forall s, t, a, x.Occurs(s, t, \text{PourWater}(a, x)) \rightarrow$$
$$\forall t'[s < t' \leq t \rightarrow Holds(t', \text{wet-infl}(a, x), \text{true})] \wedge \quad \text{(C.22)}$$
$$\forall s, t, a, x(\forall t'[s \leq t' \leq s + 3 \rightarrow (Holds(t', \text{fire-infl}(a, x), \text{true}) \wedge$$
$$Holds(t', \text{dry}(x), \text{true}) \wedge Holds(t', \text{wood}(x), \text{true}))] \rightarrow$$
$$Holds(s + 3, \text{burning}(x), \text{true})) \wedge \quad \text{(C.23)}$$
$$\forall s, a, x.\neg Holds(s, \text{dry}(x), \text{true}) \rightarrow$$
$$\neg Holds(s, \text{burning}(x), \text{true}) \wedge \quad \text{(C.24)}$$
$$\forall s, a, x.Holds(s, \text{wet-infl}(a, x), \text{true}) \rightarrow$$
$$\neg Holds(s, \text{dry}(x), \text{true}) \wedge \quad \text{(C.25)}$$
$$Holds(0, \text{dry}(\text{wood1}), \text{true}) \wedge$$
$$\neg Holds(0, \text{burning}(\text{wood1}), \text{true}) \wedge \quad \text{(C.26)}$$
$$Holds(0, \text{wood}(\text{wood1}), \text{true}) \wedge$$

$$\forall s,t,e\ [Occurs(s,t,e) \leftrightarrow \tag{C.27}$$
$$((s = 2 \wedge t = 7 \wedge e = \mathsf{LightFire}(\mathsf{bill},\mathsf{wood1})) \vee \tag{C.28}$$
$$(s = 5 \wedge t = 6 \wedge e = \mathsf{PourWater}(\mathsf{bob},\mathsf{wood1})))] \wedge \tag{C.29}$$
$$\forall t', f\ (Occlude(t', f) \leftrightarrow \exists s,t,a,x[$$
$$((s < t' \leq t \wedge f = \mathsf{fire\text{-}infl}(a,x) \wedge$$
$$Occurs(s,t,\mathsf{LightFire}(a,x))) \vee \tag{C.30}$$
$$(s < t' \leq t \wedge f = \mathsf{wet\text{-}infl}(a,x) \wedge$$
$$Occurs(s,t,\mathsf{PourWater}(a,x))) \vee \tag{C.31}$$
$$(t' = s + 3 \wedge f = \mathsf{burning}(x) \wedge \forall s'[s \leq s' \leq s + 3 \rightarrow$$
$$Holds(t', \mathsf{fire\text{-}infl}(a,x), \mathsf{true}) \wedge Holds(t', \mathsf{dry}(x), \mathsf{true})]) \vee \tag{C.32}$$
$$(f = \mathsf{burning}(x) \wedge \neg Holds(t', \mathsf{dry}(x), \mathsf{true})) \vee \tag{C.33}$$
$$(f = \mathsf{dry}(x) \wedge Holds(t', \mathsf{wet\text{-}infl}(a,x), \mathsf{true})))]) \tag{C.34}$$

An informal proof that the wood is not burning at time-point 7 can be constructed as follows. Due to C.29 and C.22 $Holds(6, \mathsf{wet\text{-}infl}, \mathsf{true})$ is true, C.33 makes sure this is legal with respect to Γ_{fl}. The value of $Holds(6, \mathsf{dry}, \mathsf{true})$ must be false due to C.25 and C.34, which means that $Holds(6, \mathsf{burning}, \mathsf{true})$ is also false due to C.24 and C.33. The only way to influence the feature burning to become true is via C.23, which states that the wood has to be dry at 3 consecutive time-points in order to burn. Furthermore, the only way that the wood can burn at time-point 7 is if it has been dry at time-points 5, 6 and 7. But we have proven that it is not dry at time-point 6, so the wood cannot be burning at time-point 7. □

Appendix D

Complete Lift Scenario

This scenario was described in Section 9.3. It uses TAL-C syntax instead of the shorthand notations presented in Chapter 7.

D.1 Scenario Setup

D.1.1 Value Sorts

`boolean`	{false, true}
`classnames`	{objectscl, liftcl, buttoncl, controllercl}
`objects`	{lifta, b0, b1, controller1}
`lift`	{lifta}
`button`	{b0, b1}
`controller`	{controller1}
`functions`	{query_floorf, init_floorf, move_upf, move_downf, press_buttonf, reset_buttonf, attach_buttonf, query_attachedf, set_liftf, serve_floorf, goto_floorf, new_goalf, query_pressedf, start_floorf}
`Integer`	{ FIXED POINT DOMAIN 'INTEGER': FROM 0 TO 10 WITH 0 DECIMALS }

subclass(`classnames`, `classnames`) : `boolean`

override(`objects`, `functions`, `classnames`) : `boolean`

D.1.2 Action Statements

$\mathbf{acs}_1$ $[t_1, t_2]$ PressButton($button$) $\rightsquigarrow$ $I([t_1]$ press_button($button$))

$\mathbf{acs}_2$ $[t_1, t_2]$ ResetButton($button$) $\rightsquigarrow$ $I([t_1]$ reset_button($button$))

$\mathbf{acs}_3$ $[t_1, t_2]$ InitializeLift($lift$, $Integer$) $\rightsquigarrow$ $I([t_1]$ init_floor($lift$, $Integer$))
$\mathbf{acs}_4$ $[t_1, t_2]$ AttachLift($controller$, $lift$) $\rightsquigarrow$ $I([t_1]$ set_lift($controller$, $lift$))
$\mathbf{acs}_5$ $[t_1, t_2]$ AttachButton($button$, $Integer$) $\rightsquigarrow$
$I([t_2]$ attach_button($button$, $Integer$))

D.1.3 Action Occurrences (initializing the problem)

$\mathbf{occ}_1$ $[0, 0]$ AttachLift(controller1, lifta)
$\mathbf{occ}_2$ $[0, 0]$ AttachButton(b0, *3*)
$\mathbf{occ}_3$ $[0, 0]$ AttachButton(b1, *5*)
$\mathbf{occ}_4$ $[0, 0]$ PressButton(b1)
$\mathbf{occ}_5$ $[0, 0]$ PressButton(b0)
$\mathbf{occ}_6$ $[0, 0]$ InitializeLift(lifta, *0*)

D.2 Classes

D.2.1 Definition of subclass

acc $[0]$subclass(sub, $super$) $\leftrightarrow$, {
$\langle sub, super\rangle = \langle$buttoncl, objectcl$\rangle \vee$
$\langle sub, super\rangle = \langle$liftcl, objectcl$\rangle \vee$
$\langle sub, super\rangle = \langle$controllercl, objectcl$\rangle$}

D.2.2 Button

attach_button(button, Integer) : boolean
attached(button) : Integer
press_button(button) : boolean
query_attached(button) : Integer
query_pressed(button) : boolean
pressed(button) : boolean
reset_button(button) : boolean
$\mathbf{dep}_{1a}$ $\forall t, classnames, button$ $[[t]$ subclass(buttoncl, $classnames$) $\rightarrow$
$I([t]$ override($button$, press_buttonf, $classnames$))]
$\mathbf{dep}_{1b}$ $\forall t, button$ $[[t]$ $\neg$override($button$, press_buttonf, buttoncl) $\wedge$
$[t]$ press_button($button$) $\rightarrow I([t]$ pressed($button$))]
$\mathbf{dep}_{2a}$ $\forall t, classnames, button$ $[[t]$ subclass(buttoncl, $classnames$) $\rightarrow$
$I([t]$ override($button$, reset_buttonf, $classnames$))]

$\mathbf{dep}_{2b}$ $\forall t, button\ [[t]\ \neg\mathsf{override}(button, \mathsf{reset\_buttonf}, \mathsf{buttoncl}) \wedge$
$[t]\ \mathsf{reset\_button}(button) \rightarrow I([t]\ \neg\mathsf{pressed}(button))]$
$\mathbf{dep}_{3a}$ $\forall t, classnames, button\ [[t]\ \mathsf{subclass}(\mathsf{buttoncl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(button, \mathsf{attach\_buttonf}, classnames))]$
$\mathbf{dep}_{3b}$ $\forall t, button, Integer\ [[t]\ \neg\mathsf{override}(button, \mathsf{attach\_buttonf}, \mathsf{buttoncl}) \wedge$
$[t]\ \mathsf{attach\_button}(button, Integer) \rightarrow I([t]\ \mathsf{attached}(button) \hat{=}$
$Integer)]$
$\mathbf{dep}_{4a}$ $\forall t, classnames, button\ [[t]\ \mathsf{subclass}(\mathsf{buttoncl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(button, \mathsf{query\_attachedf}, classnames))]$
$\mathbf{dep}_{4b}$ $\forall t, button\ [[t]\ \neg\mathsf{override}(button, \mathsf{query\_attachedf}, \mathsf{buttoncl}) \rightarrow$
$I([t]\ \mathsf{query\_attached}(button) \hat{=} value(t, \mathsf{attached}(button)))]$
$\mathbf{dep}_{5a}$ $\forall t, classnames, button\ [[t]\ \mathsf{subclass}(\mathsf{buttoncl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(button, \mathsf{query\_pressedf}, classnames))]$
$\mathbf{dep}_{5b}$ $\forall t, button\ [[t]\ \neg\mathsf{override}(button, \mathsf{query\_attachedf}, \mathsf{buttoncl}) \rightarrow$
$I([t]\ \mathsf{query\_pressed}(button) \hat{=} value(t, \mathsf{pressed}(button)))]$
$\mathbf{dep}_{6a}$ $\forall t, classnames, controller\ [[t]\ \mathsf{subclass}(\mathsf{controllercl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(controller, \mathsf{set\_liftf}, classnames))]$
$\mathbf{dep}_{6b}$ $\forall t, controller, lift\ [[t]\ \neg\mathsf{override}(controller, \mathsf{set\_liftf}, \mathsf{controllercl}) \wedge$
$[t]\ \mathsf{set\_lift}(controller, lift) \rightarrow I([t]\ \mathsf{controlled\_lift}(controller) \hat{=} lift)]$

D.2.3 Lift

currentfloor(`lift`) : `Integer`
top_floor(`lift`) : `Integer`
init_floor(`lift`, `Integer`) : `boolean`
move_down(`lift`) : `boolean`
move_up(`lift`) : `boolean`
query_floor(`lift`) : `Integer`
$\mathbf{dep}_{1a}$ $\forall t, classnames, lift\ [[t]\ \mathsf{subclass}(\mathsf{liftcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(lift, \mathsf{query\_floorf}, classnames))]$
$\mathbf{dep}_{1b}$ $\forall t, lift\ [[t]\ \neg\mathsf{override}(lift, \mathsf{query\_floorf}, \mathsf{liftcl}) \rightarrow I([t]\ \mathsf{query\_floor}(lift) \hat{=}$
$value(t, \mathsf{currentfloor}(lift)))]$
$\mathbf{dep}_{2a}$ $\forall t, classnames, lift\ [[t]\ \mathsf{subclass}(\mathsf{liftcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(lift, \mathsf{init\_floorf}, classnames))]$
$\mathbf{dep}_{2b}$ $\forall t, lift, Integer\ [[t]\ \neg\mathsf{override}(lift, \mathsf{query\_floorf}, \mathsf{liftcl}) \wedge$
$[t]\ \mathsf{init\_floor}(lift, Integer) \rightarrow I([t]\ \mathsf{currentfloor}(lift) \hat{=} Integer)]$
$\mathbf{dep}_{3a}$ $\forall t, classnames, lift\ [[t]\ \mathsf{subclass}(\mathsf{liftcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(lift, \mathsf{move\_upf}, classnames))]$

$\mathbf{dep}_{3b}$ $\forall t, lift, Integer\ [[t]\ \neg\mathsf{override}(lift, \mathsf{move\_upf}, \mathsf{liftcl}) \wedge [t]\ \mathsf{move\_up}(lift) \wedge$
$[t]\ \neg\mathsf{move\_down}(lift) \wedge [t]\ \mathsf{query\_floor}(lift) < \mathsf{top\_floor}(lift) \rightarrow$
$I([t{+}1]\ \mathsf{currentfloor}(lift) \mathrel{\hat{=}} value(t, \mathsf{currentfloor}(lift)) + 1)]$
$\mathbf{dep}_{4a}$ $\forall t, classnames, lift\ [[t]\ \mathsf{subclass}(\mathsf{liftcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(lift, \mathsf{move\_downf}, classnames))]$
$\mathbf{dep}_{4b}$ $\forall t, lift, Integer\ [[t]\ \neg\mathsf{override}(lift, \mathsf{move\_downf}, \mathsf{liftcl}) \wedge$
$[t]\ \mathsf{move\_down}(lift) \wedge [t]\ \mathsf{query\_floor}(lift) > 0 \rightarrow I([t+$
$1]\ \mathsf{currentfloor}(lift) \mathrel{\hat{=}} value(t, \mathsf{currentfloor}(lift)) - 1)]$
$\mathbf{dep}_{5a}$ $\forall t, classnames, lift\ [[t]\ \mathsf{subclass}(\mathsf{liftcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(lift, \mathsf{set\_top\_floorf}, classnames))]$
$\mathbf{dep}_{5b}$ $\forall t, lift, Integer\ [[t]\ \neg\mathsf{override}(lift, \mathsf{set\_top\_floorf}, \mathsf{liftcl}) \wedge$
$[t]\ \mathsf{set\_top\_floor}(lift, Integer) \rightarrow I([t]\ \mathsf{top\_floor}(lift) \mathrel{\hat{=}} Integer)]$

D.2.4 Controller

$\mathsf{controlled\_lift}(\mathtt{controller}) : \mathtt{lift}$
$\mathsf{set\_lift}(\mathtt{controller}, \mathtt{lift}) : \mathtt{boolean}$
$\mathsf{goal\_floor}(\mathtt{controller}) : \mathtt{Integer}$
$\mathbf{dep}_{1a}$ $\forall t, classnames, controller\ [[t]\ \mathsf{subclass}(\mathsf{controllercl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(controller, \mathsf{goto\_floorf}, classnames))]$
$\mathbf{dep}_{1b}$ $\forall t, controller, lift\ [[t]\ \neg\mathsf{override}(controller, \mathsf{goto\_floorf}, \mathsf{controllercl}) \wedge$
$[t]\ \mathsf{controlled\_lift}(controller) \mathrel{\hat{=}} lift \wedge$
$[t]\ \mathsf{query\_floor}(lift) < \mathsf{goal\_floor}(controller) \rightarrow I([t]\ \mathsf{move\_up}(lift))]$
$\mathbf{dep}_{1c}$ $\forall t, controller, lift\ [[t]\ \neg\mathsf{override}(controller, \mathsf{goto\_floorf}, \mathsf{controllercl}) \wedge$
$[t]\ \mathsf{controlled\_lift}(controller) \mathrel{\hat{=}} lift \wedge$
$[t]\ \mathsf{query\_floor}(lift) > \mathsf{goal\_floor}(controller) \rightarrow I([t]\ \mathsf{move\_down}(lift))]$
$\mathbf{dep}_{2a}$ $\forall t, classnames, controller\ [[t]\ \mathsf{subclass}(\mathsf{controllercl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(controller, \mathsf{serve\_floorf}, classnames))]$
$\mathbf{dep}_{2b}$ $\forall t, controller, button\ [t < \mathtt{maxocc} \wedge$
$[t]\ \neg\mathsf{override}(controller, \mathsf{serve\_floorf}, \mathsf{controllercl}) \wedge$
$[t]\ \mathsf{query\_floor}(value(t, \mathsf{controlled\_lift}(controller))) \mathrel{\hat{=}}$
$\mathsf{goal\_floor}(controller) \wedge [t]\ \mathsf{query\_attached}(button) \mathrel{\hat{=}}$
$\mathsf{goal\_floor}(controller) \rightarrow I([t]\ \mathsf{reset\_button}(button))]$
$\mathbf{dep}_{3a}$ $\forall t, classnames, controller\ [[t]\ \mathsf{subclass}(\mathsf{controllercl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(controller, \mathsf{new\_goalf}, classnames))]$

$\mathbf{dep}_{3b}$ $\forall t, controller\ [[t]\ \neg\mathsf{override}(controller, \mathsf{new\_goalf}, \mathsf{controllercl}) \wedge$
$[t]\ \mathsf{query\_floor}(\mathsf{lifta}) \hat{=} \mathsf{goal\_floor}(controller) \wedge$
$\exists button\ [[t]\ \mathsf{query\_pressed}(button)] \rightarrow$
$\exists button\ [[t]\ \mathsf{query\_pressed}(button) \wedge I([t+$
$1]\ \mathsf{goal\_floor}(controller) \hat{=} value(t, \mathsf{query\_attached}(button)))]]$

$\mathbf{dep}_{4a}$ $\forall t, classnames, controller\ [[t]\ \mathsf{subclass}(\mathsf{controllercl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(controller, \mathsf{start\_floorf}, classnames))]$

$\mathbf{dep}_{4b}$ $\forall controller, Integer\ [[0]\ \neg\mathsf{override}(controller, \mathsf{start\_floorf}, \mathsf{controllercl}) \wedge$
$[0]\ \mathsf{query\_floor}(\mathsf{controlled\_lift}(controller)) \hat{=} Integer \rightarrow$
$[0]\ \mathsf{goal\_floor}(controller) \hat{=} Integer]$

Appendix E

Complete Basic Cannibals and Missionaries Problem Scenario Description

This is a complete listing of the basic cannibals and missionaries problem as described in Section 9.4. This scenario and the elaborations will soon be available at the VITAL home page [55].

E.1 Scenario Setup

E.1.1 Value Sorts

`boolean`	{false, true}
`classnames`	{objectcl, vehiclecl, boatcl, groupcl, cangroupcl, misgroupcl, placecl}
`object`	{cleft, cvera, cright, mleft, mvera, mright, vera, left, right, onvera}
`vehicle`	{vera}
`boat`	{vera}
`group`	{cleft, cvera, cright, mleft, mvera, mright}
`cangroup`	{cleft, cvera, cright}
`misgroup`	{mleft, mvera, mright}
`place`	{left, right, onvera}
`bank`	{left, right}

`functions`	{ query_positionf, set_positionf, try_eatf, eat_constraintf, dief, eatf, query_sizef, move_boatf, modify_groupf, query_onboardf, boat_limitf, query_connectf, set_connectf, remove_connectf, move_personsf}
`integer`	{ FIXED POINT DOMAIN 'INTEGER': FROM -7 TO 8 WITH 0 DECIMALS }

E.1.2 Feature Symbols

subclass(`classnames`, `classnames`) : `boolean`
override(`object`, `functions`, `classnames`) : `boolean`

E.1.3 Initial State Specification

obs_1 [0] position(vera) $\hat{=}$ left
obs_2 [0] position(onvera) $\hat{=}$ nil
obs_3 [0] position(right) $\hat{=}$ nil
obs_4 [0] position(left) $\hat{=}$ nil
obs_5 [0] onboard(vera) $\hat{=}$ onvera
obs_6 [0] position(cleft) $\hat{=}$ left
obs_7 [0] position(cvera) $\hat{=}$ onvera
obs_8 [0] position(cright) $\hat{=}$ right
obs_9 [0] position(mleft) $\hat{=}$ left
obs_{10} [0] position(mright) $\hat{=}$ right
obs_{11} [0] position(mvera) $\hat{=}$ onvera
obs_{12} [0]connect$(a,b) \leftrightarrow \{\langle a,b\rangle = \langle$left, onvera$\rangle \vee \langle a,b\rangle = \langle$onvera, left$\rangle\}$
obs_{13} [0]size$(a) = b \leftrightarrow \{\langle a,b\rangle = \langle$mleft, 3$\rangle \vee \langle a,b\rangle = \langle$cleft, 3$\rangle\}$
obs_{14} [0]size$(a) = 0 \leftrightarrow \{\langle a,b\rangle \neq \langle$mleft, 3$\rangle \wedge \langle a,b\rangle \neq \langle$cleft, 3$\rangle\}$

The maximal time-point is set to 12.

`maxocc` 12

This is a VITAL feature and has nothing to do with TAL. It restricts the part of the timeline VITAL works with.

E.1.4 Goal

The goal is to have three cannibals and three missionaries on the left bank.

obs_1 [`maxocc`] query_size(mright) $\hat{=}$ *3* $\wedge$ query_size(cright) $\hat{=}$ *3*

E.1.5 Definition of subclass

acc $[0]\mathsf{subclass}(sub, super) \leftrightarrow \{$
$\langle sub, super\rangle = \langle \mathsf{boatcl}, \mathsf{objectcl}\rangle \vee$
$\langle sub, super\rangle = \langle \mathsf{placecl}, \mathsf{objectcl}\rangle \vee$
$\langle sub, super\rangle = \langle \mathsf{bankcl}, \mathsf{objectcl}\rangle \vee$
$\langle sub, super\rangle = \langle \mathsf{bankcl}, \mathsf{placecl}\rangle \vee$
$\langle sub, super\rangle = \langle \mathsf{groupcl}, \mathsf{objectcl}\rangle \vee$
$\langle sub, super\rangle = \langle \mathsf{cangroupcl}, \mathsf{groupcl}\rangle \vee$
$\langle sub, super\rangle = \langle \mathsf{cangroupcl}, \mathsf{objectcl}\rangle \vee$
$\langle sub, super\rangle = \langle \mathsf{misgroupcl}, \mathsf{groupcl}\rangle \vee$
$\langle sub, super\rangle = \langle \mathsf{misgroupcl}, \mathsf{objectcl}\rangle\}$

E.1.6 Definition of Related Groups

The cls feature is used by move_persons so that cannibals only is transfered into other cannibal groups and vice versa.

obs $[0]\mathsf{cls}(a, b) \leftrightarrow \{$
$\langle a, b\rangle = \langle \mathsf{cleft}, \mathsf{cvera}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{cleft}, \mathsf{cright}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{cvera}, \mathsf{cleft}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{cvera}, \mathsf{cright}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{cright}, \mathsf{cleft}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{cright}, \mathsf{cvera}\rangle \vee,$
$\langle a, b\rangle = \langle \mathsf{mleft}, \mathsf{mvera}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{mleft}, \mathsf{mright}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{mvera}, \mathsf{mleft}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{mvera}, \mathsf{mright}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{mright}, \mathsf{mleft}\rangle \vee$
$\langle a, b\rangle = \langle \mathsf{mright}, \mathsf{mvera}\rangle\}$

E.2 Classes

E.2.1 Object

set_position(`object`) : `place`
query_position(`object`) : `place`
position(`object`) : `place`

dep_{1a} $\forall t, classnames, object\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{objectcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(object, \mathsf{query\_positionf}, classnames))]$
dep_{1b} $\forall t, object\ [t \leq \mathtt{maxocc} \wedge [t]\ \neg\mathsf{override}(object, \mathsf{query\_positionf}, \mathsf{objectcl}) \rightarrow$
$I([t]\ \mathsf{query\_position}(object) \hat{=} value(t, \mathsf{position}(object)))]$
dep_{2a} $\forall t, classnames, object\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{objectcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(object, \mathsf{set\_positionf}, classnames))]$
dep_{2b} $\forall t, object, place\ [t \leq \mathtt{maxocc} \wedge$
$[t]\ \neg\mathsf{override}(object, \mathsf{set\_positionf}, \mathsf{objectcl}) \wedge$
$[t]\ \neg\mathsf{set\_position}(object) \hat{=} \mathsf{nil} \rightarrow$
$I([t]\ \mathsf{position}(object) \hat{=} value(t, \mathsf{set\_position}(object)))]$

E.2.2 Boat

$\mathsf{query\_onboard}(\mathtt{boat}) : \mathtt{place}$
$\mathsf{onboard}(\mathtt{boat}) : \mathtt{place}$
dep_{1a} $\forall t, classnames, boat\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{boatcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(boat, \mathsf{query\_onboardf}, classnames))]$
dep_{1b} $\forall t, boat\ [t \leq \mathtt{maxocc} \wedge [t]\ \neg\mathsf{override}(boat, \mathsf{query\_onboardf}, \mathsf{boatcl}) \rightarrow$
$I([t]\ \mathsf{query\_onboard}(boat) \hat{=} value(t, \mathsf{onboard}(boat)))]$
dep_{2a} $\forall t, classnames, boat\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{boatcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(boat, \mathsf{move\_boatf}, classnames))]$
dep_{2b} $\forall t, boat, place, place_2\ [t \leq \mathtt{maxocc} - 1 \wedge$
$[t]\ \neg\mathsf{override}(boat, \mathsf{move\_boatf}, \mathsf{boatcl}) \wedge \mathsf{query\_onboard}(boat) \hat{=} place \wedge$
$\mathsf{query\_position}(boat) \hat{=} place_2 \wedge$
$\$\mathsf{sum}(\langle group \rangle, [t]\ \mathsf{query\_position}(group) \hat{=}$
$place, value(t, \mathsf{query\_size}(group))) > 0 \rightarrow$
$\exists bank\ [value(t, \mathsf{query\_position}(boat)) \neq bank \wedge$
$I([t{+}1]\ \mathsf{set\_position}(boat) \hat{=} bank \wedge \mathsf{set\_connect}(place) \hat{=} bank \wedge$
$\mathsf{remove\_connect}(place) \hat{=} place_2)]]$
dep_{3a} $\forall t, classnames, boat\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{boatcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(boat, \mathsf{boat\_limitf}, classnames))]$
acc_{3b} $\forall t, boat, place\ [t \leq \mathtt{maxocc} \wedge ([t]\ \neg\mathsf{override}(boat, \mathsf{boat\_limitf}, \mathsf{boatcl}) \wedge$
$\mathsf{query\_onboard}(boat) \hat{=} place \rightarrow \$\mathsf{sum}(\langle group \rangle,$
$[t]\ \mathsf{query\_position}(group) \hat{=} place, value(t, \mathsf{query\_size}(group))) < 3)]$

E.2.3 Group

$\mathsf{cls}(\mathtt{object}, \mathtt{object}) : \mathtt{boolean}$
$\mathsf{modify\_group}(\mathtt{group}, \mathtt{group}) : \mathtt{integer}$

query_size(group) : integer
size(group) : integer
dep_{1a} $\forall t, classnames, group\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{groupcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(group, \mathsf{modify\_groupf}, classnames))]$
dep_{1b} $\forall t, group, integer\ [t \leq \mathtt{maxocc} \wedge$
$[t]\ \neg\mathsf{override}(group, \mathsf{modify\_groupf}, \mathsf{groupcl}) \rightarrow$
$I([t+1]\ \mathsf{size}(group) \mathrel{\hat{=}} value(t, \mathsf{size}(group)) +$
$\$\mathsf{sum}(\langle group_2 \rangle, true, value(t, \mathsf{modify\_group}(group, group_2)))$
dep_{2a} $\forall t, classnames, group\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{groupcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(group, \mathsf{query\_sizef}, classnames))]$
dep_{2b} $\forall t, group, integer\ [[t]\ \neg\mathsf{override}(group, \mathsf{query\_sizef}, \mathsf{groupcl}) \rightarrow$
$I([t]\ \mathsf{query\_size}(group) \mathrel{\hat{=}} \mathsf{size}(group))]$
dep_{3a} $\forall t, classnames, group\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{groupcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(group, \mathsf{move\_personsf}, classnames))]$
dep_{3b} $\forall t, group, group_2, integer_1, integer_2\ [$
$[t]\ \neg\mathsf{override}(group, \mathsf{move\_personsf}, \mathsf{groupcl}) \wedge \mathsf{cls}(group, group_2) \wedge$
$\mathsf{query\_size}(group) \mathrel{\hat{=}} integer_1 \wedge \mathsf{query\_size}(group_2) \mathrel{\hat{=}} integer_2 \wedge$
$[t+1]\ \mathsf{query\_connect}(\mathsf{query\_position}(group), \mathsf{query\_position}(group_2)) \rightarrow$
$\exists integer\ [-integer_2 \leq integer \leq integer_1 \wedge$
$I([t+1]\ \mathsf{modify\_group}(group, group) \mathrel{\hat{=}} -integer \wedge$
$\mathsf{modify\_group}(group_2, group) \mathrel{\hat{=}} integer)]]$
acc_4 $\forall t, group\ [[t]\ \mathsf{size}(group) \geq 0]$

E.2.4 Cannibal

dep_{1a} $\forall t, classnames, cangroup\ [t \leq \mathtt{maxocc} \wedge$
$[t]\ \mathsf{subclass}(\mathsf{cangroupcl}, classnames) \rightarrow$
$I([t]\ \mathsf{override}(cangroup, \mathsf{eat\_constraintf}, classnames))]$

acc_{1b} $\forall t, cangroup, place, integer_1, integer_2, integer_3, integer_4$ [
$[t]$ $\neg$override($cangroup$, eat_constraintf, cangroupcl) $\wedge$
query_position($cangroup$) $\hat{=}$ $place$ $\wedge$
\$sum($\langle cangroup_2 \rangle$, $[t]$ query_position($cangroup_2$) $\hat{=}$
$place$, $value(t$, query_size($cangroup_2$))) $\hat{=}$ $integer_1$ $\wedge$
\$sum($\langle boat, cangroup_2 \rangle$, $[t]$ query_position($boat$) $\hat{=}$ $place$ $\wedge$
query_onboard($boat$) $\hat{=}$
query_position($cangroup_2$), $value(t$, query_size($cangroup_2$))) $\hat{=}$ $integer_2$ $\wedge$
\$sum($\langle group \rangle$, $[t]$ query_position($group$) $\hat{=}$
$place$, $value(t$, query_size($group$))) $\hat{=}$ $integer_3$ $\wedge$
\$sum($\langle boat, group \rangle$, $[t]$ query_position($boat$) $\hat{=}$ $place$ $\wedge$
query_onboard($boat$) $\hat{=}$
query_position($group$), $value(t$, query_size($group$))) $\hat{=}$ $integer_4$ $\rightarrow$
$integer_3 + integer_4 - integer_1 - integer_2 \neq 0 \rightarrow integer_3 +$
$integer_4 \geq 2 * (integer_1 + integer_2)$)]

E.2.5 Missionary

No code needed in the basic problem.

E.2.6 Place

set_connect(`place`) : `place`
remove_connect(`place`) : `place`
query_connect(`place`,`place`) : `boolean`
connect(`place`,`place`) : `boolean`
dep_{1a} $\forall t, classnames, place$ $[t \leq$ `maxocc` $\wedge$ $[t]$ subclass(placecl, $classnames$) $\rightarrow$
$I([t]$ override($place$, query_connectf, $classnames$))]
dep_{1b} $\forall t, place, place_2$ $[t \leq$ `maxocc` $\wedge$
$[t]$ $\neg$override($place$, query_connectf, placecl) $\wedge$
connect($place, place_2$) $\rightarrow$ $I([t]$ query_connect($place, place_2$))]
dep_{2a} $\forall t, classnames, place$ $[t \leq$ `maxocc` $\wedge$ $[t]$ subclass(placecl, $classnames$) $\rightarrow$
$I([t]$ override($place$, set_connectf, $classnames$))]
dep_{2b} $\forall t, place, place_2$ $[t \leq$ `maxocc` $\wedge$
$[t]$ $\neg$override($place$, set_connectf, placecl) $\wedge$ set_connect($place$) $\hat{=}$ $place_2$ $\wedge$
$place_2 \neq$ nil $\rightarrow$ $I([t]$ connect($place, place_2$) $\wedge$ connect($place_2, place$))]
dep_{3a} $\forall t, classnames, place$ $[t \leq$ `maxocc` $\wedge$ $[t]$ subclass(placecl, $classnames$) $\rightarrow$
$I([t]$ override($place$, remove_connectf, $classnames$))]

dep_{3b} $\forall t, place, place_2\ [t \leq \mathtt{maxocc} \wedge$
$[t]\ \neg\mathsf{override}(place, \mathsf{remove\_connectf}, \mathsf{placecl}) \wedge$
$\mathsf{remove\_connect}(place) \hat{=} place_2 \wedge$
$place_2 \neq \mathsf{nil} \rightarrow I([t]\ \neg\mathsf{connect}(place, place_2) \wedge \neg\mathsf{connect}(place_2, place))]$

E.2.7 Bank

No code needed in the basic problem.

E.3 General Constraints

These constraints force at least one group to change size each time-point and that at least one group is in the boat.

acc_1 $\forall t\ [t \leq \mathtt{maxocc} - 1 \rightarrow$
$\exists group\ [[t]\ \mathsf{query\_size}(group) \hat{=} \neg value(t+1, \mathsf{query\_size}(group))]]$

acc_2 $\forall t\ [0 < t \wedge t < \mathtt{maxocc}\text{-}1 \rightarrow \$\mathsf{sum}(\langle group \rangle, [t]\ \mathsf{query\_position}(group) \hat{=}$
$\mathsf{onvera}, value(t, \mathsf{query\_size}(group))) > 0]$

Appendix F

Complete Road Network Scenario

This scenario is provided directly as written in VITAL. An overview and description is provided in Section 9.5.

F.1 Value Sorts

`boolean`	{false, true}
`classnames`	{objectcl, vehiclecl, nodecl, arccl, controllercl, drivecontrcl}
`objects`	{nil, ford, volvo, a, b, c, d, e, ab, ba, ac, ca, ad, da, cd, dc, de, ed, be, eb, vcontroller, fcontroller}
`vehicle`	{nil, ford, volvo}
`graph`	{nil, a, b, c, d, e, ab, ba, ac, ca, ad, da, cd, dc, de, ed, be, eb}
`node`	{nil, a, b, c, d, e}
`arc`	{nil, ab, ba, ac, ca, ad, da, cd, dc, de, ed, be, eb}
`controller`	{vcontroller, fcontroller}
`drivecontr`	{vcontroller}

`functions`	{ query_placef, set_placef, query_positionf, move_vehiclef, set_positionf, query_top_speedf, query_closestf, query_closest_fromf, query_closest_leavingf, query_closest_approachingf, query_green_lightf, query_distance_leavingf, query_controlsf, proceedf, drivef}
`fixpoint`	{ FIXED POINT DOMAIN 'FIXPOINT': FROM -2 TO 1000 WITH 0 DECIMALS }

F.2 Feature Symbols

connect(`graph`, `graph`) : `boolean`
controls(`controller`) : `vehicle`
distance(`arc`) : `fixpoint`
freeahead(`vehicle`) : `boolean`
move_vehicle(`vehicle`) : `fixpoint`
override(`objects`, `functions`, `classnames`) : `boolean`
place(`vehicle`) : `graph`
position(`vehicle`) : `fixpoint`
query_closest(`vehicle`) : `fixpoint`
query_closest_approaching(`node`) : `vehicle`
query_closest_from(`node`) : `vehicle`
query_closest_leaving(`node`) : `vehicle`
query_controls(`controller`) : `vehicle`
query_distance_leaving(`node`) : `fixpoint`
query_green_light(`node`) : `arc`
query_place(`vehicle`) : `graph`
query_position(`vehicle`) : `fixpoint`
query_top_speed(`vehicle`) : `fixpoint`
set_place(`vehicle`) : `graph`
set_position(`vehicle`) : `fixpoint`
speedlimit(`graph`) : `fixpoint`
subclass(`classnames`, `classnames`) : `boolean`
topspeed(`vehicle`) : `fixpoint`

F.3 Original Scenario

Scenario Description F.1
obs_1 `vardelta` = *30*

obs_2 $[0]$ $\mathsf{position}(\mathsf{volvo}) \hat{=} \mathit{460}$
obs_3 $[0]$ $\mathsf{place}(\mathsf{volvo}) \hat{=} \mathsf{ab}$
obs_4 $[0]$ $\mathsf{topspeed}(\mathsf{volvo}) \hat{=} \mathit{15}$
obs_5 $[0]$ $\mathsf{position}(\mathsf{ford}) \hat{=} \mathit{466}$
obs_6 $[0]$ $\mathsf{place}(\mathsf{ford}) \hat{=} \mathsf{eb}$
obs_7 $[0]$ $\mathsf{topspeed}(\mathsf{ford}) \hat{=} \mathit{10}$
obs_8 $[0]$ $\mathsf{topspeed}(\mathsf{nil}) \hat{=} \mathit{0}$
obs_9 $[0]$ $\mathsf{place}(\mathsf{nil}) \hat{=} \mathsf{nil}$
obs_{10} $[0]$ $\mathsf{position}(\mathsf{nil}) \hat{=} \mathit{0}$
obs_{11} $[0]$ $\mathsf{controls}(\mathsf{vcontroller}) \hat{=} \mathsf{volvo}$
obs_{12} $[0]$ $\mathsf{controls}(\mathsf{fcontroller}) \hat{=} \mathsf{ford}$
obs_1 $\$\mathsf{init}(0, \mathsf{subclass}, \mathsf{false}, \{\langle \mathsf{vehiclecl}, \mathsf{objectcl}\rangle \hat{=} \mathsf{true},$
$\langle \mathsf{nodecl}, \mathsf{objectcl}\rangle \hat{=} \mathsf{true}, \langle \mathsf{arccl}, \mathsf{objectcl}\rangle \hat{=} \mathsf{true},$
$\langle \mathsf{controllercl}, \mathsf{objectcl}\rangle \hat{=} \mathsf{true}, \langle \mathsf{drivecontrcl}, \mathsf{controllercl}\rangle \hat{=} \mathsf{true})$
// ********** Controller
// Function: query_controls()
dep_1 $\forall t, \mathit{classnames}, \mathit{controller}\ [t \leq \mathtt{maxocc} \wedge$
$[t]\ \mathsf{subclass}(\mathsf{controllercl}, \mathit{classnames}) \rightarrow I($
$[t]\ \mathsf{override}(\mathit{controller}, \mathsf{query\_controlsf}, \mathit{classnames}))]$
dep_2 $\forall t, \mathit{controller}\ [[t]\ \neg\mathsf{override}(\mathit{controller}, \mathsf{query\_controlsf}, \mathsf{controllercl}) \wedge$
$I([t]\ \mathsf{query\_controls}(\mathit{controller}) \hat{=} \mathit{value}(t, \mathsf{controls}(\mathit{controller})))]$
// Constraint: proceed()
dep_3 $\forall t, \mathit{classnames}, \mathit{controller}\ [t \leq \mathtt{maxocc} \wedge$
$[t]\ \mathsf{subclass}(\mathsf{controllercl}, \mathit{classnames}) \rightarrow$
$I([t]\ \mathsf{override}(\mathit{controller}, \mathsf{proceedf}, \mathit{classnames}))]$
dep_4 $\forall t, \mathit{controller}, \mathit{vehicle}\ [t \leq \mathtt{maxocc} - 2 \wedge$
$[t]\ \neg\mathsf{override}(\mathit{controller}, \mathsf{proceedf}, \mathsf{controllercl}) \wedge$
$\mathsf{query\_controls}(\mathit{controller}) \hat{=} \mathit{vehicle} \wedge \exists \mathit{node}\ [[t]\ \mathsf{query\_place}(\mathit{vehicle}) \hat{=}$
$\mathit{node} \wedge \$\mathsf{greater}(\mathit{value}(t, \mathsf{query\_closest}(\mathit{vehicle})), \mathtt{vardelta}) \wedge$
$\exists \mathit{arc}\ [\mathsf{connect}(\mathit{node}, \mathit{arc})]] \rightarrow \exists \mathit{arc}, \mathit{node}\ [$
$[t]\ \mathsf{query\_place}(\mathit{vehicle}) \hat{=} \mathit{node} \wedge \mathsf{connect}(\mathit{node}, \mathit{arc}) \wedge$
$I([t+1]\ \mathsf{set\_place}(\mathit{vehicle}) \hat{=} \mathit{arc})]]$
// Function: drive()
dep_5 $\forall t, \mathit{classnames}, \mathit{controller}\ [t \leq \mathtt{maxocc} \wedge$
$[t]\ \mathsf{subclass}(\mathsf{controllercl}, \mathit{classnames}) \rightarrow$
$I([t]\ \mathsf{override}(\mathit{controller}, \mathsf{drivef}, \mathit{classnames}))]$
dep_6 $\forall t, \mathit{controller}, \mathit{vehicle}\ [t \leq \mathtt{maxocc} - 2 \wedge$
$[t]\ \neg\mathsf{override}(\mathit{controller}, \mathsf{drivef}, \mathsf{controllercl}) \wedge \mathsf{query\_controls}(\mathit{controller}) \hat{=}$
$\mathit{vehicle} \rightarrow I([t]\ \mathsf{move\_vehicle}(\mathit{vehicle}) \hat{=} \mathtt{vardelta})]$

// ********** Controller elaboration: drivecontr
// Function: drive()
dep_7 $\forall t, \mathit{classnames}, \mathit{drivecontr}\ [t \leq \mathtt{maxocc} \wedge$
$[t]\ \mathsf{subclass}(\mathsf{drivecontrcl}, \mathit{classnames}) \rightarrow$
$I([t]\ \mathsf{override}(\mathit{drivecontr}, \mathsf{drivef}, \mathit{classnames}))]$
dep_8 $\forall t, \mathit{drivecontr}, \mathit{vehicle}, \mathit{arc}, \mathit{node}, \mathit{fixpoint}\ [t \leq \mathtt{maxocc} - 2 \wedge$
$[t]\ \neg\mathsf{override}(\mathit{drivecontr}, \mathsf{drivef}, \mathsf{drivecontrcl}) \wedge$
$\mathsf{query\_controls}(\mathit{drivecontr}) \hat{=} \mathit{vehicle} \wedge \mathsf{query\_place}(\mathit{vehicle}) \hat{=}$
$\mathit{arc} \wedge \$\mathsf{min}(\mathit{value}(t, \mathsf{speedlimit}(\mathit{arc})),$
$\$\mathsf{minus}(\mathit{value}(t, \mathsf{query\_closest}(\mathit{vehicle})), \mathtt{vardelta})) \hat{=} \mathit{fixpoint} \wedge$
$[t]\ \mathsf{connect}(\mathit{arc}, \mathit{node}) \wedge$
$(\$\mathsf{less}(\mathsf{query\_position}(\mathit{vehicle}), \$\mathsf{minus}(\mathsf{distance}(\mathit{arc}), \mathtt{vardelta})) \vee$
$\mathsf{query\_green\_light}(\mathit{node}) \hat{=} \mathit{arc}) \rightarrow$
$I([t]\ \mathsf{move\_vehicle}(\mathit{vehicle}) \hat{=} \mathit{fixpoint})]$
// ********** Node
// Function: query_closest_leaving()
dep_9 $\forall t, \mathit{classnames}, \mathit{node}\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{nodecl}, \mathit{classnames}) \rightarrow$
$I([t]\ \mathsf{override}(\mathit{node}, \mathsf{query\_closest\_leavingf}, \mathit{classnames}))]$
dep_{10} $\forall t, \mathit{node}\ [\exists \mathit{vehicle}\ [[t]\ \mathsf{query\_place}(\mathit{vehicle}) \hat{=} \mathit{node}] \rightarrow \exists \mathit{vehicle}\ [$
$[t]\ \mathsf{query\_place}(\mathit{vehicle}) \hat{=} \mathit{node} \wedge$
$I([t]\ \mathsf{query\_closest\_leaving}(\mathit{node}) \hat{=} \mathit{vehicle})]]$
dep_{11} $\forall t, \mathit{node}\ [\neg\exists \mathit{vehicle}\ [[t]\ \mathsf{query\_place}(\mathit{vehicle}) \hat{=} \mathit{node}] \rightarrow$
$I([t]\ \mathsf{query\_closest\_leaving}(\mathit{node}) \hat{=} \mathit{value}(t, \mathsf{query\_closest\_from}(\mathit{node})))]$
// Function: query_closest_from()
dep_{12} $\forall t, \mathit{classnames}, \mathit{node}\ [t \leq \mathtt{maxocc} \wedge [t]\ \mathsf{subclass}(\mathsf{nodecl}, \mathit{classnames}) \rightarrow$
$I([t]\ \mathsf{override}(\mathit{node}, \mathsf{query\_closest\_fromf}, \mathit{classnames}))]$
dep_{13} $\forall t, \mathit{node}\ [\exists \mathit{vehicle}_1, \mathit{arc}_1\ [\mathit{vehicle}_1 \neq \mathsf{nil} \wedge [t]\ \mathsf{connect}(\mathit{node}, \mathit{arc}_1) \wedge$
$[t]\ \mathsf{query\_place}(\mathit{vehicle}_1) \hat{=} \mathit{arc}_1 \wedge \neg\exists \mathit{vehicle}_2, \mathit{arc}_2\ [$
$[t]\ \mathsf{connect}(\mathit{node}, \mathit{arc}_2) \wedge [t]\ \mathsf{query\_place}(\mathit{vehicle}_2) \hat{=} \mathit{arc}_2 \wedge$
$[t]\ \$\mathsf{greater}(\mathsf{query\_position}(\mathit{vehicle}_1), \mathsf{query\_position}(\mathit{vehicle}_2))]] \rightarrow$
$\exists \mathit{vehicle}_1, \mathit{arc}_1\ [\mathit{vehicle}_1 \neq \mathsf{nil} \wedge [t]\ \mathsf{connect}(\mathit{node}, \mathit{arc}_1) \wedge$
$[t]\ \mathsf{query\_place}(\mathit{vehicle}_1) \hat{=} \mathit{arc}_1 \wedge \neg\exists \mathit{vehicle}_2, \mathit{arc}_2\ [$
$[t]\ \mathsf{connect}(\mathit{node}, \mathit{arc}_2) \wedge [t]\ \mathsf{query\_place}(\mathit{vehicle}_2) \hat{=} \mathit{arc}_2 \wedge$
$[t]\ \$\mathsf{greater}(\mathsf{query\_position}(\mathit{vehicle}_1), \mathsf{query\_position}(\mathit{vehicle}_2))] \wedge$
$I([t]\ \mathsf{query\_closest\_from}(\mathit{node}) \hat{=} \mathit{vehicle}_1)]]$
dep_{14} $\forall t, \mathit{node}\ [\neg\exists \mathit{vehicle}, \mathit{arc}\ [[t]\ \mathsf{connect}(\mathit{node}, \mathit{arc}) \wedge$
$[t]\ \mathsf{query\_place}(\mathit{vehicle}) \hat{=} \mathit{arc}] \rightarrow I([t]\ \mathsf{query\_closest\_from}(\mathit{node}) \hat{=} \mathsf{nil})]$
// Function: query_distance_leaving()

dep_{15} $\forall t, classnames, node\ [t \leq \mathtt{maxocc} \wedge [t]\ \ \mathsf{subclass}(\mathsf{nodecl}, classnames) \rightarrow$
$I([t]\ \ \mathsf{override}(node, \mathsf{query\_distance\_leavingf}, classnames))]$
dep_{16} $\forall t, node\ [[t]\ \ \mathsf{query\_closest\_leaving}(node) \hat{=} \mathsf{nil} \rightarrow$
$I([t]\ \ \mathsf{query\_distance\_leaving}(node) \hat{=} 999)]$
dep_{17} $\forall t, node, vehicle\ [$
$[t]\ \ \mathsf{query\_closest\_leaving}(node) \hat{=} vehicle \wedge vehicle \neq \mathsf{nil} \rightarrow$
$I([t]\ \ \mathsf{query\_distance\_leaving}(node) \hat{=} value(t, \mathsf{query\_position}(vehicle)))]$
// Function: query_closest_approaching()
dep_{18} $\forall t, classnames, node\ [t \leq \mathtt{maxocc} \wedge [t]\ \ \mathsf{subclass}(\mathsf{nodecl}, classnames) \rightarrow$
$I([t]\ \ \mathsf{override}(node, \mathsf{query\_closest\_approachingf}, classnames))]$
dep_{19} $\forall t, node\ [\exists vehicle\ [[t]\ \ \mathsf{query\_place}(vehicle) \hat{=} node] \rightarrow \exists vehicle\ [$
$[t]\ \ \mathsf{query\_place}(vehicle) \hat{=} node \wedge$
$I([t]\ \ \mathsf{query\_closest\_approaching}(node) \hat{=} vehicle)]]$
dep_{20} $\forall t, node\ [\neg\exists vehicle\ [$
$[t]\ \ \mathsf{query\_place}(vehicle) \hat{=} node] \wedge \exists vehicle_1, arc_1\ [vehicle_1 \neq \mathsf{nil} \wedge$
$[t]\ \ \mathsf{connect}(arc_1, node) \wedge$
$[t]\ \ \mathsf{query\_place}(vehicle_1) \hat{=} arc_1 \wedge \neg\exists vehicle_2, arc_2\ [$
$[t]\ \ \mathsf{connect}(arc_2, node) \wedge [t]\ \ \mathsf{query\_place}(vehicle_2) \hat{=} arc_2 \wedge$
$[t]\ \ \$\mathsf{greater}(\$\mathsf{minus}(\mathsf{distance}(arc_1), \mathsf{query\_position}(vehicle_1)),$
$\$\mathsf{minus}(\mathsf{distance}(arc_2), \mathsf{query\_position}(vehicle_2)))]] \rightarrow$
$\exists vehicle_1, arc_1\ [vehicle_1 \neq \mathsf{nil} \wedge [t]\ \ \mathsf{connect}(arc_1, node) \wedge$
$[t]\ \ \mathsf{query\_place}(vehicle_1) \hat{=} arc_1 \wedge \neg\exists vehicle_2, arc_2\ [$
$[t]\ \ \mathsf{connect}(arc_2, node) \wedge [t]\ \ \mathsf{query\_place}(vehicle_2) \hat{=} arc_2 \wedge$
$[t]\ \ \$\mathsf{greater}(\$\mathsf{minus}(\mathsf{distance}(arc_1), \mathsf{query\_position}(vehicle_1)),$
$\$\mathsf{minus}(\mathsf{distance}(arc_2), \mathsf{query\_position}(vehicle_2)))] \wedge$
$I([t]\ \ \mathsf{query\_closest\_approaching}(node) \hat{=} vehicle_1)]]$
dep_{21} $\forall t, node\ [\neg\exists vehicle\ [[t]\ \ \mathsf{query\_place}(vehicle) \hat{=} node] \wedge \neg\exists vehicle_1, arc_1\ [$
$[t]\ \ \mathsf{connect}(arc_1, node) \wedge [t]\ \ \mathsf{query\_place}(vehicle_1) \hat{=} arc_1] \rightarrow$
$I([t]\ \ \mathsf{query\_closest\_approaching}(node) \hat{=} \mathsf{nil})]$
// Function: query_green_light()
dep_{22} $\forall t, classnames, node\ [t \leq \mathtt{maxocc} \wedge [t]\ \ \mathsf{subclass}(\mathsf{nodecl}, classnames) \rightarrow$
$I([t]\ \ \mathsf{override}(node, \mathsf{query\_green\_lightf}, classnames))]$
dep_{23} $\forall t, node\ [node \neq \mathsf{nil} \wedge [t]\ \ \mathsf{query\_closest\_approaching}(node) \hat{=} \mathsf{nil} \rightarrow$
$I([t]\ \ \mathsf{query\_green\_light}(node) \hat{=} \mathsf{nil})]$
dep_{24} $\forall t, node, vehicle, arc\ [node \neq \mathsf{nil} \wedge$
$[t]\ \ \mathsf{query\_closest\_approaching}(node) \hat{=} vehicle \wedge vehicle \neq \mathsf{nil} \wedge$
$[t]\ \ \mathsf{query\_place}(vehicle) \hat{=} arc \rightarrow I([t]\ \ \mathsf{query\_green\_light}(node) \hat{=} arc)]$
// ********** Vehicle
// Function: query_place()

dep_{25} $\forall t, classnames, vehicle\ [t \leq \mathtt{maxocc} \wedge$
$[t]$ subclass(vehiclecl, $classnames$) $\rightarrow$
$I([t]$ override($vehicle$, query_placef, $classnames$))]
dep_{26} $\forall t, vehicle\ [[t]\ \neg$override($vehicle$, query_placef, vehiclecl) $\wedge$
$I([t]$ query_place($vehicle$) $\hat{=}$ $value(t,$ place($vehicle$)))]
// Procedure: set_place()
dep_{27} $\forall t, classnames, vehicle\ [t \leq \mathtt{maxocc} \wedge$
$[t]$ subclass(vehiclecl, $classnames$) $\rightarrow$
$I([t]$ override($vehicle$, set_placef, $classnames$))]
dep_{28} $\forall t, vehicle, graph\ [[t]\ \neg$override($vehicle$, set_placef, vehiclecl) $\wedge$
$[t]$ set_place($vehicle$) $\hat{=}$ $graph \wedge graph \neq$ nil $\rightarrow$
$I([t]$ place($vehicle$) $\hat{=}$ $graph$)]
// Function: query_position()
dep_{29} $\forall t, classnames, vehicle\ [t \leq \mathtt{maxocc} \wedge$
$[t]$ subclass(vehiclecl, $classnames$) $\rightarrow$
$I([t]$ override($vehicle$, query_positionf, $classnames$))]
dep_{30} $\forall t, vehicle\ [[t]\ \neg$override($vehicle$, query_positionf, vehiclecl) $\wedge$
$I([t]$ query_position($vehicle$) $\hat{=}$ $value(t,$ position($vehicle$)))]
// Procedure: set_position()
dep_{31} $\forall t, classnames, vehicle\ [t \leq \mathtt{maxocc} \wedge$
$[t]$ subclass(vehiclecl, $classnames$) $\rightarrow$
$I([t]$ override($vehicle$, set_positionf, $classnames$))]
dep_{32} $\forall t, vehicle, fixpoint\ [[t]\ \neg$override($vehicle$, set_positionf, vehiclecl) $\wedge$
$[t]$ set_position($vehicle$) $\hat{=}$ $fixpoint \wedge$ \$greater($fixpoint$, -1) $\rightarrow$
$I([t]$ position($vehicle$) $\hat{=}$ $fixpoint$)]
// Function: query_top_speed()
dep_{33} $\forall t, classnames, vehicle\ [t \leq \mathtt{maxocc} \wedge$
$[t]$ subclass(vehiclecl, $classnames$) $\rightarrow$
$I([t]$ override($vehicle$, query_top_speedf, $classnames$))]
dep_{34} $\forall t, vehicle\ [[t]\ \neg$override($vehicle$, query_top_speedf, vehiclecl) $\wedge$
$I([t]$ query_top_speed($vehicle$) $\hat{=}$ $value(t,$ topspeed($vehicle$)))]
// Procedure: move_vehicle()
dep_{35} $\forall t, classnames, vehicle\ [t \leq \mathtt{maxocc} \wedge$
$[t]$ subclass(vehiclecl, $classnames$) $\rightarrow$
$I([t]$ override($vehicle$, move_vehiclef, $classnames$))]
dep_{36} $\forall t, vehicle, fixpoint_1, fixpoint_2, fixpoint_3, arc\ [vehicle \neq$ nil $\wedge$
$[t]\ \neg$override($vehicle$, move_vehiclef, vehiclecl) $\wedge$
$[t]$ move_vehicle($vehicle$) $\hat{=}$ $fixpoint_1 \wedge$ \$greater($fixpoint_1$, 0) $\wedge$
$[t]$ query_top_speed($vehicle$) $\hat{=}$ $fixpoint_2 \wedge$

$[t]$ query_place($vehicle$) $\hat{=}$ arc $\wedge$
$[t]$ $plus(query_position($vehicle$), $min($fixpoint_1$, $fixpoint_2$)) $\hat{=}$ $fixpoint_3$ $\wedge$
$[t]$ $less($fixpoint_3$, distance($arc$)) $\rightarrow$
$I([t+1]$ set_position($vehicle$) $\hat{=}$ $fixpoint_3$)]

dep$_{37}$ $\forall t, vehicle, fixpoint_1, fixpoint_2, arc, node$ [$vehicle \neq$ nil $\wedge$
$[t]$ $\neg$override($vehicle$, move_vehiclef, vehiclecl) $\wedge$
$[t]$ move_vehicle($vehicle$) $\hat{=}$ $fixpoint_1$ $\wedge$ $greater($fixpoint_1$, 0) $\wedge$
$[t]$ query_top_speed($vehicle$) $\hat{=}$ $fixpoint_2$ $\wedge$
$[t]$ query_place($vehicle$) $\hat{=}$ arc $\wedge$ $[t]$ connect(arc, $node$) $\wedge$
$[t]$ $greaterequal($plus(query_position($vehicle$),
$min($fixpoint_1$, $fixpoint_2$)), distance(arc)) $\rightarrow$
$I([t+1]$ set_position($vehicle$) $\hat{=}$ 0 $\wedge$ set_place($vehicle$) $\hat{=}$ $node$)]

// Function: query_closest()

dep$_{38}$ $\forall t, classnames, vehicle$ [$t \leq$ maxocc $\wedge$
$[t]$ subclass(vehiclecl, $classnames$) $\rightarrow$
$I([t]$ override($vehicle$, query_closestf, $classnames$))]

dep$_{39}$ $\forall t, vehicle, arc, fixpoint$ [$t \leq$ maxocc $\wedge$ $vehicle \neq$ nil $\wedge$
$[t]$ $\neg$override($vehicle$, query_closestf, vehiclecl) $\wedge$
$[t]$ query_place($vehicle$) $\hat{=}$ arc $\wedge$ $\exists vehicle_2$ [
$[t]$ query_place($vehicle_2$) $\hat{=}$ arc $\wedge$
$[t]$ $less(query_position($vehicle$), query_position($vehicle_2$)) $\wedge$ $\neg\exists vehicle_3$ [
$[t]$ query_place($vehicle_3$) $\hat{=}$ arc $\wedge$
$[t]$ $less(query_position($vehicle_3$), query_position($vehicle_2$)) $\wedge$
$[t]$ $less(query_position($vehicle$), query_position($vehicle_3$))]] $\rightarrow$
$\exists vehicle_2$ [[t] query_place($vehicle_2$) $\hat{=}$ arc $\wedge$
$[t]$ $less(query_position($vehicle$), query_position($vehicle_2$)) $\wedge$ $\neg\exists vehicle_3$ [
$[t]$ query_place($vehicle_3$) $\hat{=}$ arc $\wedge$
$[t]$ $less(query_position($vehicle_3$), query_position($vehicle_2$)) $\wedge$
$[t]$ $less(query_position($vehicle$), query_position($vehicle_3$))] $\wedge$
$I([t]$ query_closest($vehicle$) $\hat{=}$
$value(t, $minus(query_position($vehicle_2$), query_position($vehicle$))) $\wedge$
freeahead($vehicle$) $\hat{=}$ false)]]

dep$_{40}$ $\forall t, vehicle, arc, node$ [$t \leq$ maxocc $\wedge$ $vehicle \neq$ nil $\wedge$
$[t]$ $\neg$override($vehicle$, query_closestf, vehiclecl) $\wedge$
$[t]$ query_place($vehicle$) $\hat{=}$ arc $\wedge$ freeahead($vehicle$) $\wedge$ connect(arc, $node$) $\wedge$
query_closest_leaving($node$) $\hat{=}$ nil $\rightarrow$ $I([t]$ query_closest($vehicle$) $\hat{=}$ 999)]

dep$_{41}$ $\forall t, vehicle, arc, node, vehicle_4$ [$t \leq$ maxocc $\wedge$ $vehicle \neq$ nil $\wedge$
$[t]$ $\neg$override($vehicle$, query_closestf, vehiclecl) $\wedge$
$[t]$ query_place($vehicle$) $\hat{=}$ arc $\wedge$ $[t]$ freeahead($vehicle$) $\wedge$

$[t]$ connect$(arc, node) \wedge$
$[t]$ query_closest_leaving$(node) \hat{=} vehicle_4 \wedge vehicle_4 \neq$ nil $\rightarrow$
$I([t]$ query_closest$(vehicle) \hat{=}$
$value(t,$ \$minus(\$plus(query_position$(vehicle_4)$, distance$(arc))$,
query_position$(vehicle))))]$

dep$_{42}$ $\forall t, vehicle, node, vehicle_4$ $[t \leq$ maxocc $\wedge$ $vehicle \neq$ nil $\wedge$
$[t]$ $\neg$override$(vehicle,$ query_closestf, vehiclecl$) \wedge$
$[t]$ query_place$(vehicle) \hat{=} node \wedge [t]$ query_closest_from$(node) \hat{=}$ nil $\rightarrow$
$I([t]$ query_closest$(vehicle) \hat{=} 999)]$

dep$_{43}$ $\forall t, vehicle, node, vehicle_4$ $[t \leq$ maxocc $\wedge$ $vehicle \neq$ nil $\wedge$
$[t]$ $\neg$override$(vehicle,$ query_closestf, vehiclecl$) \wedge$
$[t]$ query_place$(vehicle) \hat{=} node \wedge$
$[t]$ query_closest_from$(node) \hat{=} vehicle_4 \wedge vehicle_4 \neq$ nil $\rightarrow$
$I([t]$ query_closest$(vehicle) \hat{=} value(t,$ query_position$(vehicle_4)))]$

Bibliography

[1] The Java homepage http://java.sun.com.

[2] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[3] J F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, july 1984.

[4] James F. Allen. Temporal reasoning and planning. In Allen, Kautz, Pelavin, and Tenenberg, editors, *Reasoning About Plans*. Morgan Kaufmann, 1991. ISBN 1-55860-137-6.

[5] E. Amir. Object-oriented first-order logic. Workshop on Nonmonotonic Reasoning, Action and Change. IJCAI 99, Aug 1999.

[6] A. Artale and E. Franconi. A temporal description logic for reasoning about actions and plans. *Journal of Artificial Intelligence Research*, Vol 9:463–506, 1998.

[7] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000. Available at ftp://newlogos.uwaterloo.ca/pub/bacchus/BKTIplan.ps.

[8] A B. Baker. A simple solution to the Yale shooting problem. In Ronald J Brachman, Hector J Levesque, and Raymond Reiter, editors, *Proceedings of the Firstst International Conference on Principles on Knowledge Representation and Reasoning (KR-89)*, pages 11–20, Toronto, ON, Canada, May 1989. Morgan Kaufmann.

[9] C. Baral and M. Gelfond. Representing concurrent actions in extended logic programming. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 866–871, Chambery, France, 1993. Morgan Kaufmann.

[10] K. Belleghem, M. Denecker, and D. Dupré. Representing ramifications in an event-based language. Technical report CW 257, Department of Computer Science, K.U. Leuven, 1997.

[11] G. Booch. *Object-Oriented Design with Applications.* The Benjamin/ Cummings Publishing Company, Inc, 1991.

[12] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick. CLASSIC: A structural data model for objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland Oregon, May-June 1989.

[13] S. Bornscheuer and M. Thielscher. Explicit and implicit indeterminism: Reasoning about uncertain and contradictory specifications of dynamic systems. *Journal of Logic Programming. Special issue on reasoning about action and change*, 31(1–3):119–155, 1997.

[14] R. Brachman, R. Fikes, and H. Levesque. KRYPTON: A functional approach to knowledge representation. *Computer*, 16:67–73, 1983.

[15] K. Clark. Negation as failure. *Logic and Data Bases*, pages 293–322, 1978.

[16] J. de Kleer and J.S. Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[17] M. Denecker, D. Dupré, and K. Belleghem. An inductive definition approach to ramifications. *Linköping Electronic Articles in Computer and Information Science*, 3(007), 1998. Available at http:// www.ep.liu.se/ea/cis/1998/007/.

[18] J. Doherty, P. Gustafsson. Delayed effects of actions = direct effects + causal rules. *Linköping Electronic Articles in Computer and Information Science*, 1998. Available on WWW: http://www.ep.liu.se/ea/cis/1998/001.

[19] P. Doherty. Notes on PMON circumscription. Technical report, Department of Computer and Information Science, Linköping University, 1994. Available on WWW: http://www.ida.liu.se/labs/rkllab/people/patdo/.

[20] P. Doherty. Reasoning about action and change using occlusion. In *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 401–405. John Wiley & Sons, 1994.

[21] P. Doherty. PMON$^+$: A fluent logic for action and change. Technical Report 96-33, Department of Computer and Information Science, Linköping University, 1996. Available on WWW: http://www.ida.liu.se/ patdo.

[22] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnström. TAL: Temporal action logics language specification and tutorial. Linköping Electronic Articles in Computer and Information Science, 1998. Available at: http://www.ep.liu.se/ea/cis/1998/015/.

[23] P. Doherty and J. Kvarnström. Tackling the qualification problem using fluent dependency constraints: Preliminary report. In *Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning (TIME'98)*, Sanibel Island, Florida, USA, May 1998. IEEE Computer Society Press. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/time98.ps.gz.

[24] P. Doherty and J. Kvarnström. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In C. Dixon and M. Fisher, editors, *Proceedings of the Sixth International Workshop on Temporal Representation and Reasoning (TIME'99)*, pages 47–54, Orlando, Florida, USA, May 1999. IEEE Computer Society Press. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/time99-final.ps.gz.

[25] P. Doherty and W. Lukaszewicz. Circumscribing features and fluents. In *Proceedings of the 1st International Conference on Temporal Reasoning*, pages 82–100. Springer, 1994.

[26] P. Doherty, W. Lukaszewicz, and A. Szalas. Computing circumscription revisited: A reduction algorithm. *Journal of Automated Reasoning*, 18:297–336, 1996.

[27] P. Doherty and P. Peppas. A comparison between two approaches to ramification: PMON(R) and AR_0. In *Proceedings of the eighth Australian Joint Conference on Artificial Intelligence*, pages 267–274, 1995.

[28] C. Elkan. Reasoning about actions in first-order logic. In *Proceedings of the Conference of the Canadian Society for Computational Studies of Intelligence*, Vancouver, Canada, 1992. Morgan Kaufmann.

[29] J. Ferber and J. Müller. Influences and reaction: a model of situated multi-agent systems. In Mario Tokoro, editor, *Proceedings of the Second International Conference on Multi-Agent Systems*, Kyoto, December 1996. AAAI Press.

[30] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[31] J.J. Finger. *Exploiting constraints in design synthesis.* PhD thesis, Department of Computer Science, Stanford University, 1987.

[32] K.D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[33] D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic, Mathematical Foundations and Computational Aspects*, volume 1. Oxford University Press, 1994.

[34] H. Geffner. Causality, constraints and the side effects of actions. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Naogoya Japan, 1997.

[35] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, pages 17:301–322, 1993.

[36] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus? In *Working notes, AAAI Spring Symposium, Series. Symposium: Logical Formalization of Commonsense Reasoning*, pages 59–69, Stanford California, 1991.

[37] M. Georgeff. Actions, processes, and causality. In M. Georgeff and Lansky A., editors, *Reasoning about actions and plans: Proceedings of the 1986 workshop.* Morgan Kaufmann, 1987.

[38] M.L. Ginsberg and D.E Smith. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35:165–195, 1988.

[39] M.L. Ginsberg and D.E Smith. Reasoning about action II: The qualification problem. *Artificial Intelligence*, 35:311–342, 1988.

[40] P. Grünwald. Causation, explanation and nonmonotonic temporal reasoning. Technical Report INS-R9701, Centrum voor Wiskunde en Informatica, January 1997.

[41] J. Gustafsson. Object-oriented reasoning about action and change. In *Proceedings of the Seventh Scandinavian Conference on Artificial Intelligence*, feb 2001.

[42] J. Gustafsson and P. Doherty. Embracing occlusion in specifying the indirect effects of actions. *Proceedings of the international Conference on principles of knowledge representation and reasoning*, pages 87–98, 1996.

[43] J. Gustafsson and J. Kvarnström. Breaking causal cycles. Unpublished.

[44] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence, 33(3)*, 33:379–412, 1987.

[45] G. Hendrix. Modeling simultaneous actions and continous processes. *Artificial Intelligence*, 4:145–180, 1973.

[46] A. Henschel and M. Thielscher. The LMW traffic world in the fluent calculus. Linköping University Electronic Press. http://www.ida.liu.se/ext/epa/cis/lmw/001/tcover.html, 1999.

[47] L. Karlsson and J. Gustafsson. Reasoning about actions in a multi-agent environment. *Linköping University Electronic Press*, 2(014), 1997. Available at http://www.ep.liu.se/ea/cis/1997/014.

[48] L. Karlsson and J. Gustafsson. Reasoning about concurrent interaction. *Journal of Logic and Computation*, 9(5):623–650, October 1999.

[49] L. Karlsson, J. Gustafsson, and P. Doherty. Delayed effects of actions. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence*, pages 542–546, Aug 1998.

[50] G.N. Kartha and V. Lifschitz. Actions with indirect effects: Preliminary report. *International Conference on Knowledge Representation and Reasoning*, pages 341–350, 1994.

[51] M. Koubarakis. Complexity results for first-order theories of temporal constraints. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference*, pages 379–390, San Francisco, California, May 1994. Morgan Kaufmann.

[52] J. Kvarnström and P. Doherty. Tackling the qualification problem using fluent dependency constraints. *Computational Intelligence*, 16(2):169–209, May 2000.

[53] J. Kvarnström and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 2001. Accepted for publication.

[54] J. Kvarnström, P. Doherty, and P. Haslum. Extending TALplanner with concurrency and resources. In W. Horn, editor, *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-2000)*, pages 501–505, Berlin, Germany, August 2000. IOS Press, The Netherlands. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/www-ecai.ps.gz.

[55] J. Kvarnström and P. Doherty. VITAL. An on-line system for reasoning about action and change using TAL, 1997–2000. Available at http:www.ida.liu.se/~jonkv/vital.html.

[56] A. Lansky. A representation of parallel activity based on events, structure, and causality. In M. Georgeff and A. Lansky, editors, *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 123–159, Los Altos, California, 1987. Morgan Kaufmann.

[57] H. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Electronic Transactions on Artificial Intelligence*, 2:159–178, 1998. http://www.ep.liu.se/ej/etai/1998/005/.

[58] H Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. Journal of Logic Programming 31:59– 84., 1997.

[59] R. Li and L.M. Pereira. Temporal reasoning and abductive logic programming. In *Proceedings of the Twelfth European Conference on Artificial Intelligence*, pages 13–17. John Wiley & Sons, 1996.

[60] V. Lifschitz. Computing circumscription. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 121–127, Los Angeles, California, 1985. Morgan Kaufmann.

[61] V. Lifschitz. Pointwise circumscription. In M. Ginsberg, editor, *Readings in Non-monotonic Reasoning*. Morgan Kauffmann, Los Altos, California, 1987.

[62] V. Lifschitz. Frames in the space of situations. *Artificial Intelligence*, 46:365–376, 1990.

[63] V. Lifschitz. Circumscription. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Nonmonotonic Reasoning and Uncertain Reasoning*, volume 3 of *Handbook of Artificial Intelligence and Logic Programming*, pages 179–193. Oxford University Press, 1994.

[64] V. Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96:451–465, 1997.

[65] V. Lifschitz. Missionaries and cannibals in the causal calculator. In *"Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000)"*, pages 85–96, April 2000.

[66] V. Lifschitz and A. Rabinov. Things that change by themselves. In *Proceedings of Eleventh International Joint Conference on Artificial Intelligence*, pages 864–867, Detroit, MI, 1989.

[67] F. Lin. Embracing causality in specifying the indirect effects of actions. *Proceedings of Fourteenth International Joint Conference on Artificial Intelligence*, pages 1985–1991, 1995.

[68] F. Lin. Embracing causality in specifying the indeterminate effects of actions. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*. AAAI Press, Menlo Park, California, 1996.

[69] F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation, Special Issue on Actions and Processes, 4(5)*, pages 655–678, October 1994.

[70] F. Lin and Y. Shoham. Provably correct theories of actions. *Journal of ACM*, 42(2):293–320, 1995.

[71] H. McCain, N. Turner. A causal theory of ramifications and qualifications. *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1978–1984, 1995.

[72] H. McCain, N. Turner. Causal theories of action and change. *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 460–465, 1997.

[73] N. McCain and the Texas Action Group. The causal calculator. http://www.cs.utexas.edu/users/tag/cc/.

[74] J McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[75] J. McCarthy. Elaboration tolerance. In *Common Sense 98*, London, Jan 1998.

[76] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[77] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Sciences*, 6:101–155, 1982.

[78] L. Morgenstern. Inheritance comes of age: Applying nonmonotonic techniques to problems in industry. *Artificial Intelligence*, 103:1–34, 1998.

[79] C. Moss. *Prolog++, The power of object-oriented and logic programming.* Addison-Wesley, 1994.

[80] J. Pearl. Embracing causality in default reasoning. *Artificial Intelligence*, 35:259–271, 1988.

[81] J. Pearl. Causation, action and counterfactuals. *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge*, pages 51–73, March 1996.

[82] R. Pelavin. Planning with simultaneous actions and external events. In Allen, Kautz, Pelavin, and Tenenberg, editors, *Reasoning About Plans*, pages 127–212. Morgan Kaufmann, San Mateo, California, 1991.

[83] J. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, January 1994.

[84] J. Pinto. Concurrent actions and interacting effects. In *Proceedings of Knowledge Representation and Reasoning Conference*, pages 292–303, San Francisco, California, 1998.

[85] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–360. Academic Press, San Diego, California, 1991.

[86] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference*, pages 2–13. Morgan Kaufmann, 1996.

[87] R. Reiter. Sequential, temporal GOLOG. In *Proceedings of the Sixth International Conference on Principles on Knowledge Representation and Reasoning (KR-98)*, pages 2–5, Trent, Italy, June 1998. Morgan Kaufmann.

[88] E. Sandewall. Logic modelling workshop: Communicating axiomatizations of actions and change. http://www.ida.liu.se/ext/etai/lmw.

[89] E. Sandewall. Filter preferential entailment for the logic of action and change. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, (IJCAI-89)*, pages 894–899. Morgan Kaufmann, 1989.

[90] E. Sandewall. *Features and Fluents*, volume 1. Oxford Press, 1994.

[91] E. Sandewall. Assessments of ramification methods that use static domain constraints. *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

[92] M. Shanahan. Representing continuous change in the situation calculus. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pages 598–603. John Wiley & Sons, 1990.

[93] M. Shanahan. *Solving the Frame Problem : A mathematical investigation of the common sense law of inertia.* MIT Press, London, 1997.

[94] M. Shanahan. The ramification problem in event calculus. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 140–146, 1999.

[95] Y. Shoham. Chronological ignorance: Time, nonmonotonicity, necessity and causal theories. *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 389–393, 1986.

[96] Y. Shoham. *Reasoning about Change.* MIT Press, London, 1987.

[97] M. Thielscher. The logic of dynamic systems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1956–1962, Montreal, Canada, Aug 1994. Morgan Kaufmann.

[98] M. Thielscher. Computing Ramifications by Postprocessing. In C. S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1994–2000, Montreal, Canada, August 1995. Morgan Kaufmann.

[99] M. Thielscher. Ramification and causality. *Artificial Intelligence*, 89:317–364, 1997.

[100] M. Thielscher. Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence*, 3(014), 1998. http://www.ep.liu.se/ea/cis/1998/014/.

[101] M. Winslett. Reasoning about actions using a possible models approach. *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 89–93, 1988.

[102] C. Yi. Towards assessment of logics for concurrent actions. In *Proceedings of FAPR'96: International Conference on Formal and Applied Practical Reasoning*, pages 679–690, Bonn, Germany, June 1996. Springer.