Elaboration Tolerance through Object-Orientation

Joakim Gustafsson and Jonas Kvarnström
Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden
{joagu,jonkv}@ida.liu.se

Abstract

Although many formalisms for reasoning about action and change have been proposed in the literature, their semantic adequacy has primarily been tested using tiny domains that highlight some particular aspect or problem. However, since some of the classical problems are completely or partially solved and since powerful tools are available, it is now necessary to start modeling more complex domains. This paper presents a methodology for handling such domains in a systematic manner using an object-oriented framework and provides several examples of the elaboration tolerance exhibited by the resulting models.

1 Introduction

Traditionally, the semantic adequacy of formalisms for reasoning about action and change (RAC) has primarily been tested using tiny specialized domains that highlight some particular point an author wants to make. These domains can usually be represented as a small number of simple formulas that are normally grouped by type rather than structure.

However, with some of the classical RAC problems completely or partially solved, and with powerful tools available for reasoning about action scenarios, it is now possible to model larger and more realistic domains. As soon as we start doing this, it becomes apparent that there is an unfortunate lack of methodology for handling complex domains in a systematic manner. There are few (if any) principles of good form, like the "No Structure in Function" principle from the qualitative reasoning community [7].

The following are some questions that must be answered in order to develop such a methodology:

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?

Elaboration tolerance [17]: How do we ensure that domains 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?

In this paper, we investigate the applicability of the object-oriented paradigm [1, 4] to answering these questions. Using the order-sorted logic TAL-C [11] as a basis, we model the entities that appear in a domain as *objects*, encapsulated abstractions that offer a well-defined *interface* to the surrounding world and hide the implementation-specific details. The interface consists of *methods* that can be called by other objects. Objects are instances of *classes* sharing the same attributes and methods. Classes are ordered in an *inheritance hierarchy* where a class can be created as a subclass of another class, inheriting its attributes and methods and possibly adding its own or redefining inherited methods.

Modeling entities as objects and interacting with them using methods provides a high degree of consistency in the domain model. The fact that attributes are hidden and accessed using methods increases elaboration tolerance, and modularity and reusability are aided by modeling self-contained classes that are independent of the implementations of other classes.

In the remainder of this paper, we will introduce TAL-C (Section 2), show how domains can be modeled in TAL-C in an object-oriented manner (Section 3), present a model of the Missionaries and Cannibals domain (Section 4) and some elaborations of this domain (Section 5), briefly mention the Traffic World domain (Section 6), and finally conclude with related work (Section 7) and a discussion of the results (Section 8).

2 The TAL-C Logic

TAL-C [11] is a member of the TAL (Temporal Action Logics [9]) family of logics. The basic approach for reasoning about action and change in TAL is as follows.

First, represent a narrative in the surface language $\mathcal{L}$(ND), a high-level macro language for representing observations, action descriptions, action occurrences, domain constraints, and dependency constraints (causal constraints).

Second, translate the narrative into the base language $\mathcal{L}$(FL), an order-sorted first-order language with four predicates:

- $\mathit{Holds}(t, f, v)$ – the fluent f takes on the value v at time t,
- $\mathit{Occlude}(t, f)$ – f is exempt from the default assumptions at t (see below),
- $\mathit{Per}(f)$ – f is persistent: Unless occluded at $t + 1$, it must retain the value it had at time t,
- $\mathit{Dur}(f, v)$ – f is durational: Unless occluded at t, it must take on its default value v.

A linear discrete time structure is used. The minimization policy is based on the use of filtered preferential entailment [21] where action descriptions and dependency constraints are circumscribed with *Occlude* minimized and all other predicates fixed; due to structural constraints on $\mathcal{L}$(ND) statements, this circumscription is equivalent to using a predicate completion procedure. The result is filtered with two nochange axioms, the observations and domain constraints, and some foundational axioms such as unique names, domain closure (all value domains are finite) and temporal structure axioms. The second-order theory can be translated into a logically equivalent first-order theory which is used to reason about the narrative.

The translation from $\mathcal{L}$(ND) to $\mathcal{L}$(FL) is straightforward and the reader is referred to [9, 11] for details.

2.1 Macros in $\mathcal{L}$(ND)

Since the purpose of this paper is not to extend TAL but to show how the object-oriented paradigm can be used to succinctly structure large axiomatic theories, only those $\mathcal{L}$(ND) macros that will be used in the examples will be described.

A *fixed fluent formula* has the form $[t] f \hat{=} v$ and is true iff the fluent f has the value v at time t. For boolean fluents, the shorthand notation $[t] f$ or $[t] \neg f$ is allowed. The function $\mathit{value}(t, f)$ denotes the value of f at t.

Fluent values are changed using the **Set** operator, which has also been denoted I (interval reassignment) in some articles. The formula $\mathbf{Set}([t] f \hat{=} v)$, translated into $\mathit{Holds}(t, f, v) \wedge \mathit{Occlude}(t, f, v)$, means that f must take on the value v at time t and that the fluent is exempt from the persistence or default value assumption at that timepoint.

For both types of formulas, boolean connectives are allowed within the temporal scope. The notation is extended for open, closed and semi-open intervals.

Formulas in $\mathcal{L}$(ND) are formed from these expressions in a manner similar to the definition of well-formed formulas in a first-order logical language using the standard connectives, quantifiers and notational conventions. Free variables are assumed to be implicitly universally quantified.

3 Object-Oriented Modeling in TAL

As has been shown previously [11, 12, 13], TAL is a flexible and fine-grained logic suitable for handling a wide class of domains. We will now show how to use object-oriented modeling as a structuring mechanism for domain descriptions, thereby supporting the modeling of more complex domains and increasing the possibility of being able to reuse existing models when modeling related domains.

To simplify the task of the domain designer, some extensions to the $\mathcal{L}$(ND) syntax will be introduced. These extensions are not essential: The new macros and statement classes can mechanically be translated into the older syntax. We provide informal definitions in this article, and the complete translations will be presented in a forthcoming technical report.

3.1 Defining Classes and Attributes

In TAL, domains are traditionally modeled using a set of boolean or non-boolean fluents, each of which can take a number of arguments belonging to specific value domains. Each value domain is specified by explicitly enumerating its members.

In the object-oriented approach, we will instead concentrate on *classes* and *objects*. Each class will be modeled as a finite value domain, and each object as a value in that domain. Due to the order-sorted type structure used in TAL, inheritance hierarchies for classes are easily supported by modeling subclasses as subdomains. We will assume that the hierarchy has a single root called OBJECT.

Attributes (fields) associated with objects of a specific class are specified in attribute declarations, and may have arguments. An attribute $\mathsf{a}(s_1, \ldots, s_n)$ of type s in a class c is translated into a fluent $\mathsf{a}(c, s_1, \ldots, s_n) : s$ taking an additional argument of type c.

As an example, consider a simple water tank domain. Any water tank has a volume, a maximum volume and a base area, all of which are `Real` values:

dom OBJECT
dom TANK *extends* OBJECT
attr TANK.volume, TANK.maxvol : `Real`

This is translated into the value domains TANK and OBJECT and the two fluents volume(TANK) : `Real` and maxvol(TANK) : `Real`.

We define the new syntax $obj.\mathsf{attr}(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \mathsf{attr}(obj, x_1, \ldots, x_n)$, where $n \geq 0$; if $n = 0$, the parentheses may be omitted.

Subclasses inherit the attributes of their parents (for example, the argument to the volume fluent can belong to a subdomain of TANK), and can add a new set of attributes. A tank with a flow in or out of the tank can be modeled as follows:

dom FLOWTANK *extends* TANK
attr FLOWTANK.flow : `Real`

3.2 Creating Objects

Objects are declared using object statements. Declaring an object as a member of a class c naturally also makes it a member of its superclasses. The mechanically generated domain closure axioms ensure that no objects exist except those explicitly defined using object statements.

obj **tank1**: TANK
obj **tank2**, **tank3**: FLOWTANK

Note that since classes correspond to value domains, it is possible to quantify over all objects belonging to a given class.

Attributes are initialized at time 0 using ordinary logic formulas (observation statements). This also allows the use of partially specified attributes:

obs $\forall tank.[0]\ tank.\mathsf{volume} \hat{=} 0$
obs $\forall flowtank.[0]\ flowtank.\mathsf{flow} \leq 2$

3.3 Methods

In a classical object-oriented view, 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 formulas that must be satisfied whenever the method is invoked. Methods can be invoked over intervals of time, and several methods can be invoked concurrently.

Three different kinds of methods are defined: Mutators (which are called in order to change the state of an object), constraint methods (which are not explicitly invoked but must hold at all timepoints), and accessors (which query the state of an object).

Mutators can be called to change the internal state of an object, and are modeled as dependency constraints triggered by *invocation fluents*.

To define a mutator method with $n \geq 0$ arguments of sorts $\langle s_1, \ldots, s_n \rangle$ in class CLASS, we first define a boolean durational invocation fluent $\mathsf{method}(\text{CLASS}, s_1, \ldots, s_n)$ with default value false. The method implementation consists of one or more dependency constraints where the precondition contains $\mathsf{method}(\text{CLASS}, s_1, \ldots, s_n)$. To call the method for the object obj with the actual arguments $x_1, \ldots, x_n$ at time t, we make the invocation fluent $\mathsf{method}(obj, x_1, \ldots, x_n)$ true at t.

In order to make the syntax more similar to that of ordinary object-oriented languages, a method call macro $\mathbf{Call}(\tau, f) \stackrel{\text{def}}{=} \mathbf{Set}([\tau]f \hat{=} \top)$ is introduced and we define $obj.\mathsf{method}(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \mathsf{method}(obj, x_1, \ldots, x_n)$; if $n = 0$, the parentheses may be omitted.

For example, we can define a mutator set_volume(`Real`) in class TANK and set the volume of **tank1** to 4.5 at time 2 as follows:

dep $\forall t, tank \in \text{TANK}, f \in \texttt{Real}$
$[t]\,tank.\mathsf{set\_volume}(f) \rightarrow \mathbf{Set}([t]\,tank.\mathsf{volume} \hat{=} f)$
dep $\mathbf{Call}(2, \mathbf{tank1}.\mathsf{set\_volume}(4.5))$

Constraint methods model behaviors that should always be active. Instead of being triggered by invocation fluents, constraint methods are active at all timepoints. In a sense, they could be viewed as mutators that are continuously invoked. This allows many common RAC constructions such as state constraints to be expressed while keeping an object-oriented viewpoint.

The fact that the volume of water in a FLOWTANK changes according to the flow of water can be encoded as follows:

dep $\mathbf{Set}([t+1]\,tank.\mathsf{volume} \hat{=} value(t, tank.\mathsf{volume} + tank.\mathsf{flow}))$

Accessors are used for querying the state of an object, either by retrieving the current value of an attribute or by performing complex calculations.

Accessors are modeled using *return value fluents* that should take on the desired return value at all timepoints. Like constraint methods, accessors do not need to be triggered but are active at all timepoints. For example, a simple query_volume() method for a water tank can be modeled by introducing a persistent fluent query_volume(TANK) : `Real` and adding the following dependency constraint:

dep $\mathbf{Set}([t]\ tank.\mathsf{query\_volume}() \hat{=} value(t, tank.\mathsf{volume})$

A slightly more complex accessor might determine whether the tank is full:

dep $\mathbf{Set}([t]\ tank.\mathsf{query\_full}() \leftrightarrow value(t, tank.\mathsf{volume}) = value(t, tank.\mathsf{maxvol}))$

3.4 Explicit Class Structure

In some cases, for example when overriding method implementations (Section 3.5.1), there are advantages to allowing logic formulas to directly inspect the class structure.

Since TAL has no built-in means for doing this, we mechanically construct a value domain classname containing all class names before the translation from $\mathcal{L}$(ND) to $\mathcal{L}$(FL). We also declare and initialize a boolean fluent[1] subclass(classname, classname), where subclass(c_1, c_2) is true iff c_1 is a subclass of c_2. For the water tank example, the definitions would be equivalent to the following:

$$\begin{array}{ll}
\mathsf{dom} & \mathtt{classname} = \{\textsc{object}, \textsc{tank}, \textsc{flowtank}\} \\
\mathsf{acc} & \forall t, c_1 \in \mathtt{classname}, c_2 \in \mathtt{classname} \\
& \quad \mathsf{subclass}(c_1, c_2) \leftrightarrow \\
& \quad\; ((c_1 = \textsc{flowtank} \wedge c_2 = \textsc{object}) \vee \\
& \quad\;\; (c_1 = \textsc{flowtank} \wedge c_2 = \textsc{tank}) \vee \\
& \quad\;\; (c_1 = \textsc{tank} \wedge c_2 = \textsc{object}))
\end{array}$$

3.5 Elaborating a Domain

There are numerous ways of elaborating an existing domain definition.

A new top-level class or subclass can be added using a dom statement. This requires trivial changes to the automatically generated classname domain and subclass fluent.

Also, any class can easily be extended with new attributes and methods without the need to modify the existing parts of the class definition. Adding new methods may yield a new definition of the automatically generated *Occlude* predicate (the TAL approach to solving the frame problem). However, the new definition can be created by analyzing the new methods in isolation and adding new disjuncts to the existing definition of *Occlude*.

Finally, methods implemented in a superclass can be overridden (redefined) in a subclass.

3.5.1 Overriding Method Implementations

The fact that the implementation of a method method in class classname is overridden for an object *object*, due to *object* belonging to a subclass where method is overridden, is explicitly modeled using the boolean fluent override(*object*, method, classname). This fluent is durational with default value false: Overriding only occurs where explicitly forced.

Method implementations should be conditionalized on not being overridden, and methods should explicitly override implementations in superclasses. The former is achieved by adding a suitable override expression in the precondition of the method. For example, we could modify the set_volume mutator as follows:

$$\begin{array}{ll}
\mathsf{dep} & \forall t, tank \in \textsc{tank}, f \in \mathtt{Real} \\
& \quad [t]\, tank.\mathsf{set_volume}(f) \wedge \\
& \qquad \neg\mathsf{override}(tank, \mathsf{set_volume}, \textsc{tank}) \rightarrow \\
& \quad \mathbf{Set}([t]\, tank.\mathsf{volume} \mathrel{\hat{=}} f)
\end{array}$$

The latter is done by adding a statement of the following form each time a method methodname is defined in a class CURRENTCLASS:

$$\begin{array}{ll}
\mathsf{dep} & \forall t, c \in \mathtt{classnames}, i \in \textsc{currentclass} \\
& \quad [t]\mathsf{subclass}(\textsc{currentclass}, c) \rightarrow \\
& \quad \mathbf{Set}([t]\mathsf{override}(i, \mathsf{methodname}, c)),
\end{array}$$

where i ranges over all instances of CURRENTCLASS. For convenience, the macro **ClassMethod**(CURRENTCLASS, methodname) will be used as a shorthand for statements of this type.

3.6 Type Identification

A constraint method query_type(): classname is introduced in the root class OBJECT to support type identification. During the translation process, an implementation is generated for every class CLASS as follows:

$$\begin{array}{ll}
\mathsf{dep} & \mathbf{ClassMethod}(\textsc{class}, \mathsf{query_type}) \\
\mathsf{dep} & \forall t, self \in \textsc{class} \\
& \quad [t]\neg\mathsf{override}(self, \mathsf{query_type}, \textsc{class}) \rightarrow \\
& \qquad self.\mathsf{query_type}() \mathrel{\hat{=}} \textsc{class}
\end{array}$$

4 Missionaries and Cannibals

McCarthy [17] illustrates his ideas regarding elaboration tolerance with 19 elaborations of the Missionaries and Cannibals Problem (MCP). We will now model the basic, unelaborated problem using the object-oriented constructions presented above. As we will show in the next section, the ability to override methods and to add new methods and attributes also provides a natural way to model many of the elaborations.

4.1 Overview of the Design

The basic version of the MCP is as follows:

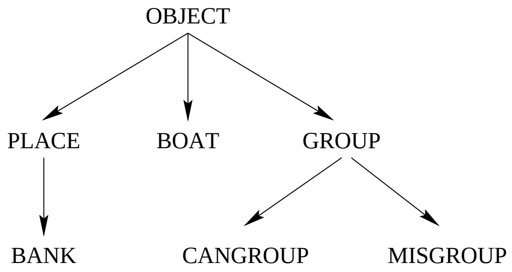
> Three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross in order to avoid anyone being eaten?

Although we know we will eventually need to model some elaborated versions of the domain, we will attempt to ignore that knowledge and provide a model suitable for this particular version of the MCP. This will provide a

[1] Although we do not intend to change subclass relations over time, TAL has no support for time-independent functions.

better test for whether the object-oriented model is truly elaboration tolerant.

We will define classes for objects, boats, places, and banks. Like Lifschitz [15], we will model missionaries and cannibals as *groups* of a certain size rather than as individuals. In the standard domain, there will be six (possibly empty) groups: Missionaries and cannibals at the left bank, at the right bank, and on the boat.



Although one could use the model only for prediction and then apply standard planning algorithms to solve each problem, we instead model the possible choice of actions at each timepoint within the logic, using incompletely specified constraint methods: Whenever there are people on board a boat, it will automatically move to another (unspecified) bank, and whenever a boat is at a bank, an unspecified number of missionaries and cannibals (possibly zero) will move between the two places.

The state at time 0 is constrained to be the initial state, where everyone is at the left bank. Due to the incompleteness in the constraint methods, there will generally be an infinite number of logical models where people and boats move in various ways.

An additional constraint method ensures that the cannibals never outnumber the missionaries in any location.

Finally, to find a plan that moves everyone to the right bank within the given constraints, we assume (like Lifschitz [15]) that we know the length l of the plan to be generated. By constraining the state at time l to be a solution state, we ensure that any logical model must correspond to a valid plan.[2]

4.2 Object

The root class OBJECT has a pos attribute representing its location:

dom OBJECT
attr OBJECT.pos: PLACE

The following methods are available:

Mutator set_pos(PLACE): Sets the position of the object.

[2] Note that this procedure depends on the fact that all incomplete information corresponds to possible choices of actions rather than incomplete knowledge about the world.

dep **ClassMethod**(OBJECT, set_pos)
dep $[t]\neg$override(*object*, set_pos, OBJECT)$\wedge$
object.set_pos(*place*) $\to$
Set($[t]$*object*.pos $\hat{=}$ *place*)

Accessor query_pos(): Returns the position of the object.

dep **ClassMethod**(OBJECT, query_pos)
dep $[t]\neg$override(*object*, query_pos, OBJECT) $\to$
Set($[t]$*object*.query_pos() $\hat{=}$ *value*(t, *object*.pos))

In the remainder of this paper, attributes will generally be assumed to have accessors and mutators following this pattern.

4.3 Place

A PLACE may be connected to other places. This is represented using a boolean attribute connection with a PLACE argument.

dom PLACE *extends* OBJECT
attr PLACE.connection(PLACE) : boolean

The following methods are added:

Mutator add_connection(PLACE_2): Connects this PLACE to PLACE_2.

Mutator remove_connection(PLACE_2): Removes the connection to PLACE_2.

Accessor query_connection(PLACE_2): Returns true if this PLACE is connected to PLACE_2.

4.4 Bank

A BANK is a PLACE where a boat can be located. The standard MCP has two banks: The left bank and the right bank.

dom BANK *extends* PLACE

4.5 Group

A GROUP represents a group of people in a certain location; subclasses such as CANGROUP and MISGROUP will be used for specific types of people. It adds two new methods and a size attribute specifying the number of people in the group.

dom GROUP *extends* OBJECT
attr GROUP.size : Integer

Mutator modify_group(GROUP_2,*Integer*): The first argument specifies who is modifying the group, which is necessary to allow multiple groups to add or remove people from this group concurrently. The second argument specifies the number of people being added or removed.

This method does not follow the standard pattern where each invocation triggers a separate rule. Instead, a single rule sums the arguments of all concurrent invocations:[3]

dep $\quad$ $\textbf{ClassMethod}(\textsc{group}, \mathsf{modify\_group})$

dep $\quad$ $[t]\neg\mathsf{override}(\mathit{group}, \mathsf{modify\_group}, \textsc{group}) \rightarrow$

$$\textbf{Set}([t+1]\mathsf{size}(\mathit{group}) \hat{=} \mathit{value}(t, \mathsf{size}(\mathit{group})) + \sum_{\{\langle g,x\rangle \ | \ g\in\textsc{group}\wedge[t] \ \mathit{group}.\mathsf{modify\_group}(g,x)\}} x$$

Constraint move_persons(): Moves an unspecified number of people (possibly zero) between compatible groups in connected locations. Group objects are assumed to be compatible iff they belong to the same class; group classes for missionary groups and cannibal groups are defined below. For example, if there is a group of cannibals g_1 on the left bank and a group of cannibals g_2 on the boat, and the boat is at the left bank (the places are connected), then cannibals may move between g_1 and g_2. Note that GROUPs never move – people move by changing the size of two groups.

The exact number of people moved by this method will be constrained indirectly by the goal as described in Section 4.1.

dep $\quad$ $\textbf{ClassMethod}(\textsc{group}, \mathsf{move\_persons})$

dep $\quad$ $[t]\neg\mathsf{override}(g_1, \mathsf{move\_persons}, \textsc{group}) \wedge$
$g_1.\mathsf{query\_type}() \hat{=} g_2.\mathsf{query\_type}() \wedge$
$[t+1]g_1.\mathsf{query\_pos}().\mathsf{query\_connection}(g_2.\mathsf{query\_pos}()) \rightarrow$
$\exists \mathit{Integer}\ [-\mathit{value}(t, g_2.\mathsf{query\_size}()) \leq \mathit{Integer} \wedge$
$\mathit{Integer} \leq \mathit{value}(t, g_1.\mathsf{query\_size}()) \wedge$
$\textbf{Call}(t+1, g_1.\mathsf{modify\_group}(g_2, -\mathit{Integer})) \wedge$
$\textbf{Call}(t+1, g_2.\mathsf{modify\_group}(g_1, \mathit{Integer}))]$

The macro $\textbf{people\_at}(\tau, \textsc{group}, \mathit{place})$ will denote the number of people at *place* that belong to the given group GROUP at time τ:

$$\textbf{people\_at}(\tau, \textsc{group}, \mathit{place}) = \sum_{\{g \ | \ g\in\textsc{group}\wedge[\tau]g.\mathsf{query\_pos}()\hat{=}\mathit{place}\}} \mathit{value}(\tau, g.\mathsf{query\_size}())$$

For example, given that **left** denotes the left bank, $\textbf{people\_at}(7, \textsc{cangroup}, \textbf{left})$ denotes the number of cannibals on the left bank at time 7.

4.6 Cannibals

A CANGROUP is a group of cannibals.

dom $\quad$ CANGROUP *extends* GROUP

Constraint eat_constraint(): Specifies that there cannot be more cannibals than missionaries at any place.

[3]Throughout this paper we will use summation over a set as a shorthand. Since TAL uses finite domains, each expression can be rewritten as a finite expression using plain addition.

4.7 Missionaries

A MISGROUP is a group of missionaries. The class extends GROUP and adds no new methods or attributes.

dom $\quad$ MISGROUP *extends* GROUP

4.8 Boat

A BOAT is used to cross the river. Its onboard attribute points to the PLACE onboard the boat (the pos of any GROUP onboard the boat).

dom $\quad$ BOAT *extends* OBJECT

attr $\quad$ BOAT.onboard : PLACE

There are two methods:

Constraint boat_limit(): There must never be more than two passengers.

Constraint move_boat(): If anybody is onboard a boat, the boat automatically moves to another (unspecified) BANK. The destination bank is unspecified, and will be constrained indirectly by the goal as described in Section 4.1.

dep $\quad$ $\textbf{ClassMethod}(\textsc{boat}, \mathsf{move\_boat})$

dep $\quad$ $[t]\neg\mathsf{override}(\mathit{boat}, \mathsf{move\_boat}, \textsc{boat}) \wedge$
$\mathit{boat}.\mathsf{query\_onboard}() \hat{=} \mathit{place} \wedge$
$\textbf{people\_at}(t, \textsc{group}, \mathit{place}) > 0 \rightarrow$
$\exists \mathit{bank}'\ [\mathit{bank}' \neq \mathit{value}(t, \mathit{boat}.\mathsf{query\_pos}()) \wedge$
$\textbf{Call}(t+1, \mathit{boat}.\mathsf{set\_pos}(\mathit{bank}')) \wedge$
$\textbf{Call}(t+1, \mathit{place}.\mathsf{add\_connection}(\mathit{bank}')) \wedge$
$\textbf{Call}(t+1, \mathit{place}.\mathsf{remove\_connection}(\mathit{bank}))]$

4.9 General Constraints

Apart from the classes and methods specified above, there are also two constraints that prune uninteresting state sequences. First, we should never be idle – at each timepoint, at least one group should change sizes. Second, there should be at least one person on the boat, except at the first and last timepoint. These constraints sometimes provide a considerable increase in speed when searching for a solution.

4.10 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 $\quad$ **left**,**right** : BANK

obj $\quad$ **onvera** : PLACE

obj $\quad$ **vera** : BOAT

obj $\quad$ **cleft**,**cvera**,**cright** : CANGROUP

obj $\quad$ **mleft**,**mvera**,**mright** : MISGROUP

The following observation statements specify the attributes of these objects:

obs $[0]\mathbf{vera}.\mathsf{pos} \hat{=} \mathbf{left} \land \mathbf{vera}.\mathsf{onboard} \hat{=} \mathbf{onvera}$
obs $[0]\mathbf{cleft}.\mathsf{pos} \hat{=} \mathbf{left} \land \mathbf{cleft}.\mathsf{size} \hat{=} 3$
obs $[0]\mathbf{cvera}.\mathsf{pos} \hat{=} \mathbf{onvera}$
obs $[0]\mathbf{cright}.\mathsf{pos} \hat{=} \mathbf{right}$
obs $[0]\mathbf{mleft}.\mathsf{pos} \hat{=} \mathbf{left} \land \mathbf{mleft}.\mathsf{size} \hat{=} 3$
obs $[0]\mathbf{mvera}.\mathsf{pos} \hat{=} \mathbf{onvera}$
obs $[0]\mathbf{mright}.\mathsf{pos} \hat{=} \mathbf{right}$
acc $[0]\mathit{group}.\mathsf{size} \hat{=} 0 \leftrightarrow (\mathit{group} \neq \mathbf{mleft} \land \mathit{group} \neq \mathbf{cleft})$
acc $[0]\mathit{place}_1.\mathsf{connect}(\mathit{place}_2) \leftrightarrow$
$((\mathit{place}_1 = \mathbf{left} \land \mathit{place}_2 = \mathbf{onvera}) \lor$
$(\mathit{place}_1 = \mathbf{onvera} \land \mathit{place}_2 = \mathbf{left}))$

Finally, a goal is required. We know that the minimal plan length is 12:

obs $[12]\mathbf{mright}.\mathsf{size} \hat{=} 3 \land \mathbf{cright}.\mathsf{size} \hat{=} 3$

5 Elaborations

According to McCarthy [17], elaboration tolerance is "the ability to accept changes to a person's or a computer program's representation of facts about a subject without having to start all over".

Several ideas used in the object-oriented paradigm facilitate the creation of elaboration tolerant domain models. This is not surprising, since the reasons behind the object-oriented paradigm include modularization and the possibility to reuse code. Inheritance makes it possible to specialize a class, adding new attributes, methods and constraints. Using overriding, the behaviors of a superclass can be changed without knowing implementation-specific details and without the need for "surgery" (McCarthy's term for modifying a domain description by actually changing or removing formulas or terms rather than merely adding facts).

Using the object-oriented model of the MCP domain defined above as a basis, we have modeled the 19 elaborations defined by McCarthy; the complete elaborations will soon be available at the VITAL web page [14]. It should be mentioned that some of the elaborations are rather vaguely formulated, and we do not claim to have captured every aspect of each problem or that the formalism always allows the elaborations to be expressed as succinctly as possible. However, we do feel that the main points have been modeled in a reasonable manner.

Below, we show how to model a subset of the elaborations. The timings were generated by the research tool VITAL [14] on a 440 MHz UltraSparc machine. We also provide some comparisons with the 10 elaborations implemented by Lifschitz [15] in the Causal Calculator [16], which was run on an unspecified machine. The timings are *not* directly comparable and should not be taken as claims regarding the efficiency of the two approaches.

5.1 Domain and Problem Specifications

We will consider each problem to consist of two parts: The *domain specification*, which defines the classes being used together with their attributes and the inheritance hierarchy, and the *problem specification*, which defines the object instances being used in a specific problem instance together with the initial values of their attributes.

Our focus has been on elaboration tolerance for the domain specification. Each elaboration may add new classes, or add new methods or attributes to existing classes. Note that no part of the original $\mathcal{L}$(ND) domain specification is removed or modified except the definitions of the `classname` domain and the `subclass` fluent.

Although it would have been possible to use similar techniques to model the problem specification in Section 4.10 in a defeasible manner, 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 places are connected) are generated from scratch each time. The problem instance definitions for the elaborations below are generally trivial and will usually be omitted.

5.2 The Original Problem

The original problem is solved in 2 seconds by VITAL.

5.3 The Boat is a Rowboat (#1)

The fact that the boat is a rowboat can be modeled by making **vera** an instance of a new class ROWBOAT. The problem is still solved in 2 seconds.

dom ROWBOAT *extends* BOAT
obj **vera** : ROWBOAT

5.4 Hats (#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 PERSON.hat : HAT
attr GROUP.contains(PERSON) : `boolean`

The contains attribute is non-inert.

Nobody belongs to two groups, and everybody belongs to a group:

dep $\textbf{ClassMethod}(\textsc{person}, \mathsf{unique})$
acc $[t]\neg\mathsf{override}(\mathit{person}, \mathsf{unique}, \textsc{person}) \wedge$
$\mathit{group}_1.\mathsf{query\_contains}(\mathit{person}) \wedge$
$\mathit{group}_2.\mathsf{query\_contains}(\mathit{person}) \rightarrow \mathit{group}_1 = \mathit{group}_2$
dep $\textbf{ClassMethod}(\textsc{person}, \mathsf{belongs})$
acc $[t]\neg\mathsf{override}(\mathit{person}, \mathsf{belongs}, \textsc{person}) \rightarrow$
$\exists \mathit{group}.[t]\mathit{group}.\mathsf{contains}(\mathit{person})$

Finally, an additional rule is added for modify_group: If g_1 moves n of its people to another group g_2, then there should be exactly n individual PERSONs that used to belong to g_1 but now belong to g_2.

acc $[t]\neg\mathsf{override}(g_1, \mathsf{modify\_group}, \textsc{group}) \wedge$
$[t+1]g_1.\mathsf{modify\_group}(g_2, \mathit{Integer}) \wedge$
$\mathit{Integer} \geq 0 \wedge g_1 \neq g_2 \rightarrow$
$\sum_{\{p \ | \ p \in \textsc{person} \wedge [t]g_1.\mathsf{contains}(p) \wedge [t+1]g_2.\mathsf{contains}(p)\}} 1 = \mathit{Integer}$

This problem is solved (without exchanging any hats) in 50 seconds.

5.5 Four of Each (#3)

There are four missionaries and four cannibals. In our terminology, this is a change in the problem specification rather than in the domain specification. We therefore modify the problem specification accordingly. The problem is unsolvable, which is detected by VITAL in 26 seconds.

5.6 The Boat Can Carry Three (#4)

There are four missionaries and four cannibals. 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 deliberately 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 SIZEBOAT.capacity : Integer
dep $\textbf{ClassMethod}(\textsc{sizeboat}, \mathsf{boat\_limit})$
acc $[t]\neg\mathsf{override}(\mathit{sizeboat}, \mathsf{boat\_limit}, \textsc{sizeboat}) \rightarrow$
$\textbf{people\_at}(t, \textsc{group}, \mathit{value}(t, \mathit{sizeboat}.\mathsf{query\_onboard}())) \leq$
$\mathit{value}(t, \mathit{sizeboat}.\mathsf{capacity})$

A solution is found in 15 seconds (18 for Lifschitz).

5.7 Not Everybody Can Row (#6 and #7)

In elaborations 6 and 7, not everybody can row. Two new classes for rowing cannibals and rowing missionaries are introduced:

dom ROWCANGROUP *extends* CANGROUP
dom ROWMISGROUP *extends* MISGROUP

The new constraint method BOAT.row_limit() ensures that no boat moves unless there is someone aboard who can row.

dep $\textbf{ClassMethod}(\textsc{boat}, \mathsf{row\_limit})$
acc $[t]\neg\mathsf{override}(\mathit{boat}, \mathsf{row\_limit}, \textsc{boat}) \wedge$
$\mathit{boat}.\mathsf{query\_pos}() \not\doteq \mathit{value}(t+1, \mathit{boat}.\mathsf{query\_pos}()) \wedge$
$\mathit{boat}.\mathsf{query\_onboard}() \mathrel{\hat{=}} \mathit{place} \rightarrow$
$\textbf{people\_at}(t, \textsc{rowcan}, \mathit{place}) +$
$\textbf{people\_at}(t, \textsc{rowmis}, \mathit{place}) > 0$

Elaboration 6: Only one cannibal and one missionary can row. Apart from the additions to the domain specification, the problem specification must be modified so that one rowing and two non-rowing cannibals are created, in addition to one rowing and two non-rowing missionaries. This problem is solved in 27 seconds (compared to Lifschitz' 273 seconds).

Elaboration 7: No missionary can row. The problem is unsolvable, which is detected in 2.5 seconds.

5.8 Big Cannibal, Small Missionary (#9)

There is a big cannibal and a small missionary. The big cannibal can eat the small missionary if they are alone in the same place.

We add the classes SMALLMISGROUP for small missionaries and BIGCANGROUP for large cannibals together with a constraint method eat_small that ensures that a small missionary and a big cannibal are never isolated together.

dom SMALLMISGROUP *extends* MISGROUP
dom BIGCANGROUP *extends* CANGROUP
dep $\textbf{ClassMethod}(\textsc{bigcangroup}, \mathsf{eat\_small})$
acc $[t]\neg\mathsf{override}(\mathit{bigcangroup}, \mathsf{eat\_small}, \textsc{bigcangroup}) \wedge$
$\textbf{people\_at}(t, \textsc{bigcangroup}, \mathit{place}) = 1 \wedge$
$\textbf{people\_at}(t, \textsc{smallmisgroup}, \mathit{place}) = 1 \rightarrow$
$\textbf{people\_at}(t, \textsc{group}, \mathit{place}) > 2$

With this addition, the problem is solved in 12 steps in 209 seconds (compared to Lifschitz' 22 seconds).

5.9 Jesus (#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 4.5 is then overridden with a variation where the condition $[t+1]\mathit{group}.\mathsf{query\_pos}().\mathsf{query\_connection}(\mathit{group}_2.\mathsf{query\_pos}())$ 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 in 4 seconds.

5.10 Conversion (#11)

Three missionaries can convert an isolated cannibal. Add a constraint method convert in class MISGROUP:

dep $\textbf{ClassMethod}(\text{MISGROUP}, \mathsf{convert})$
dep $[t]\neg\mathsf{override}(\mathit{misgroup}, \mathsf{convert}, \text{MISGROUP}) \wedge$
$\textbf{people\_at}(t, \text{MISGROUP}, \mathit{place}) \geq 3 \wedge$
$\textbf{people\_at}(t, \text{CANGROUP}, \mathit{place}) = 1 \rightarrow$
$\textbf{Call}(t+1, \mathit{misgroup}.\mathsf{modify\_group}(\mathit{misgroup}, 1)) \wedge$
$\textbf{Call}(t+1, \mathit{misgroup}.\mathsf{modify\_group}(\mathit{cangroup}, -1))$

This elaboration takes advantage of the true concurrency in TAL [11]. For example, modify_group automatically handles situations where a cannibal is boarding a boat while another is being converted to a missionary.

This problem is solved in 3 seconds. This cannot be compared to Lifschitz' solution, which does not permit this kind of concurrency.

5.11 The Boat Might Be Stolen (#12)

Whenever a cannibal is alone in a 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 timepoint of a model is the probability of that particular plan succeeding.

attr $\text{BOAT}.\mathsf{prob\_not\_stolen} : \texttt{Real}$
obs $\forall \mathit{boat}.[0]\,\mathit{boat}.\mathsf{prob\_not\_stolen} \hat{=} 1.0$
dep $\textbf{ClassMethod}(\text{BOAT}, \mathsf{update\_prob})$
dep $[t]\neg\mathsf{override}(\mathit{boat}, \mathsf{update\_prob}, \text{BOAT}) \wedge$
$\mathit{boat}.\mathsf{query\_onboard}() \hat{=} \mathit{place} \wedge$
$\textbf{people\_at}(t, \text{GROUP}, \mathit{place}) = 1 \wedge$
$\textbf{people\_at}(t, \text{CANGROUP}, \mathit{place}) = 1 \rightarrow$
$\textbf{Set}([t+1]\mathit{boat}.\mathsf{prob\_not\_stolen} \hat{=} 0.9 *$
$\mathit{value}(t, \mathit{boat}.\mathsf{prob\_not\_stolen}))$

A plan is found in 7 seconds.

5.12 The Bridge (#13)

There is a bridge, where two people can cross concurrently. Add a BRIDGE class and ensure that at most two people can be on any bridge:

dom BRIDGE *extends* PLACE
dep $\textbf{ClassMethod}(\text{BRIDGE}, \mathsf{bridge\_limit})$
acc $[t]\neg\mathsf{override}(\mathit{bridge}, \mathsf{bridge\_limit}, \text{BRIDGE}) \rightarrow$
$\textbf{people\_at}(t, \text{GROUP}, \mathit{bridge}) \leq 2$

Then instantiate a bridge and connect it to the left and right banks. This problem is solved in 2.5 seconds and requires 5 steps. Since Lifschitz does not allow the use of the bridge and the boat concurrently, the solutions cannot be compared.

5.13 The Boat Can Be Damaged (#15)

The boat may suffer damage and have to be taken back to the left side for repairs. 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 some time on the river before arriving.

dom SLOWBOAT *extends* BOAT
attr $\text{SLOWBOAT}.\mathsf{emergency} : \text{BOOLEAN}$
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* timepoints 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 [8, 12]

dep $\textbf{ClassMethod}(\text{SLOWBOAT}, \mathsf{move\_boat})$
dep $[t]\neg\mathsf{override}(\mathit{slowboat}, \mathsf{move\_boat}, \text{SLOWBOAT}) \wedge$
$\mathit{slowboat}.\mathsf{query\_onboard}() \hat{=} \mathit{place}_1 \wedge$
$\mathit{slowboat}.\mathsf{query\_pos}() \hat{=} \mathit{place}_2 \wedge$
$\mathit{place}_2 \neq \mathit{onriver} \wedge$
$\textbf{people\_at}(t, \text{GROUP}, \mathit{place}_1) > 0 \rightarrow$
$\textbf{Call}(t+1, \mathit{slowboat}.\mathsf{set\_pos}(\textbf{onriver})) \wedge$
$\textbf{Call}(t+1, \mathit{place}.\mathsf{remove\_connection}(\mathit{place}_2))$
dep $[t]\neg\mathsf{override}(\mathit{slowboat}, \mathsf{move\_boat}, \text{SLOWBOAT}) \wedge$
$\mathit{slowboat}.\mathsf{query\_onboard}() \hat{=} \mathit{place} \wedge$
$\mathit{slowboat}.\mathsf{query\_pos}() \hat{\neq} \textbf{onriver} \wedge$
$(t, t+\mathit{crosstime}]\ \mathit{slowboat}.\mathsf{query\_pos}() \hat{=} \textbf{onriver} \wedge$
$(t, t+\mathit{crosstime}]\ \mathit{slowboat}.\mathsf{query\_emergency}() \hat{=} \bot \rightarrow$
$\exists \mathit{bank}[[t]\ \mathit{slowboat}.\mathsf{query\_pos}() \hat{\neq} \mathit{bank} \wedge$
$\textbf{Call}(t+\mathit{crosstime}, \mathit{slowboat}.\mathsf{set\_pos}(\mathit{bank})) \wedge$
$\textbf{Call}(t+\mathit{crosstime}, \mathit{place}.\mathsf{add\_connection}(\mathit{bank}))]$

If there is an emergency, the boat should move to the left bank and be repaired:

dep $\textbf{ClassMethod}(\text{SLOWBOAT}, \mathsf{emergency\_behavior})$
dep $[t]\neg\mathsf{override}(\mathit{slowboat}, \mathsf{emergency\_behavior}, \text{SLOWBOAT}) \wedge$
$\mathit{slowboat}.\mathsf{query\_emergency}() \wedge$
$\mathit{slowboat}.\mathsf{query\_onboard}() \hat{=} \mathit{place} \rightarrow$
$\textbf{Call}(t+3, \mathit{slowboat}.\mathsf{set\_pos}(\textbf{left})) \wedge$
$\textbf{Call}(t+3, \mathit{place}.\mathsf{add\_connection}(\textbf{left})) \wedge$
$\textbf{Call}(t+3, \mathit{slowboat}.\mathsf{set\_emergency}(\bot))]$

If $\mathit{crosstime} = 3$ and the boat breaks once, at time 20, this problem is solved in 49 seconds.

5.14 The Island (#16)

If an island is added, the problem can be solved with four missionaries and four cannibals. Change the number of people initially present on the left bank and add an island object:

obj **island** *instanceof* BANK

The problem is solved in 4862 seconds (compared to 1894 seconds for Lifschitz' partial solution where three missionaries and three cannibals end up on the right bank).

5.15 Two Sets of People (#19)

There are two sets of missionaries and cannibals too far apart along the river to interact. A new attribute connected keeps track of which banks are connected:

attr $\textsc{bank}.\mathsf{connected}(\textsc{bank}) : \mathtt{boolean}$

Any BOAT moves nondeterministically between all banks. The constraint method move_connected ensures that the origin and destination are connected.

dep $\mathbf{ClassMethod}(\textsc{boat}, \mathsf{move\_connected})$
dep $[t]\neg\mathsf{override}(boat, \mathsf{move\_connected}, \textsc{boat}) \rightarrow$
$\quad boat.\mathsf{query\_pos}().\mathsf{query\_connected}($
$\qquad value(t + 1, boat.\mathsf{query\_pos}()))$

This problem is solved in 16 seconds.

6 Traffic World

The object-oriented framework presented in this paper has also been used for modeling the Traffic World scenario proposed in the Logic Modeling Workshop [20], previously modeled by Henschel and Thielscher [10] using the Fluent Calculus [22]. This domain consists of cars moving in a road network represented as a graph structure, together with a TAL controller class that "drives" a car. A complete TAL action scenario will soon be available at the VITAL web page [14].

7 Related Work

Much work has been done in combining ideas found in object-oriented languages with the area of knowledge representation. One such area is description logics [5, 6], 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 seen as sub-languages of predicate logic. Starting with primitive concepts and roles, one 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.

Description logic hierarchies are very dynamic, and 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 [3].

The modeling methodology presented in this paper uses a simpler class hierarchy that is fixed at translation time. Classes are explicitly positioned in the hierarchy, and classes and objects cannot be constructed once the narrative has been translated. Also, description logics do not use 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++ [19], C++ or Java. In most such languages, however, a method is a sequence of code that is procedurally executed when the method is invoked. In our approach, a method is a set of rules that must be satisfied whenever the method is invoked. Since delays can be modeled in TAL, methods can be invoked over intervals of time and 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 [2], 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 [18] 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.

8 Discussion

This paper has presented a way to do object-oriented modeling in an existing logic of action and change. This allows large domains to be modeled in a more systematic way and provides 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 timeline 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 timepoints to complete.

Although a number of new macros have been introduced in this paper, those macros are merely syntactic sugar serving to simplify the construction of domain descriptions. Thus, the most important contribution is not the syntax but the structure that is enforced on standard TAL-C narratives to improve modularity and reusability. It is also reasonable to believe that the added structure could be used to make theorem proving in $\mathcal{L}$(FL) more efficient, although the current version of VITAL does not take advantage of this.

9 Acknowledgements

This research is supported in part by the Swedish Research Council for Engineering Sciences (TFR), the WITAS Project under the Wallenberg Foundation and the ECSEL/ENSYM graduate studies program.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[2] E. Amir. Object-oriented first-order logic. Linköping Electronic Articles in Computer and Information Science, 2001.

[3] 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.

[4] G. Booch. *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc, 1991.

[5] 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.

[6] R. Brachman, R. Fikes, and H. Levesque. KRYPTON: A functional approach to knowledge representation. *Computer*, 16:67–73, 1983.

[7] J. de Kleer and J. S. Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[8] P. Doherty and J. 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.

[9] 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*, 3(15), September 1998. Available at http://www.ep.liu.se/ea/cis/1998/015.

[10] 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.

[11] L. Karlsson and J. Gustafsson. Reasoning about concurrent interaction. *Journal of Logic and Computation*, 9(5):623–650, October 1999.

[12] 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.

[13] J. Kvarnström and P. Doherty. Tackling the qualification problem using fluent dependency constraints. *Computational Intelligence*, 16(2):169–209, May 2000.

[14] 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.

[15] 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.

[16] N. McCain and the Texas Action Group. The causal calculator. http://www.cs.utexas.edu/users/tag/cc/.

[17] J. McCarthy. Elaboration tolerance. In *Common Sense 98*, London, Jan 1998.

[18] L. Morgenstern. Inheritance comes of age: Applying nonmonotonic techniques to problems in industry. *Artificial Intelligence*, 103:1–34, 1998.

[19] C. Moss. *Prolog++, The power of object-oriented and logic programming*. Addison-Wesley, 1994.

[20] E. Sandewall. Logic modelling workshop: Communicating axiomatizations of actions and change. http://www.ida.liu.se/ext/etai/lmw.

[21] 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.

[22] M. Thielscher. Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence*, 3(014), 1998. http://www.ep.liu.se/ea/cis/1998/014/.