Elaboration Tolerance through Object-Orientation

Joakim Gustafsson [a], Jonas Kvarnström *

[a] *Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden*

Abstract

Although many formalisms for reasoning about action and change have been proposed in the literature, any concrete examples provided in such articles have primarily consisted of tiny domains that highlight some particular aspect or problem. However, since some of the classical problems are now completely or partially solved and since powerful tools are becoming available, it is now necessary to start modeling more complex domains. This article 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 very small 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

* Corresponding author. E-mail address: jonkv@ida.liu.se.
Email addresses: joakim.gustafsson@rmrocade.com (Joakim Gustafsson), jonkv@ida.liu.se (Jonas Kvarnström).
URLs: http://www.ida.liu.se/~joagu (Joakim Gustafsson), http://www.ida.liu.se/~jonkv (Jonas Kvarnström).

Accepted for publication in Artificial Intelligence Journal, 2003

form, like the "No Structure in Function" principle from the qualitative reasoning community [1].

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 several developers to work on the same domain description and to enable others to understand the resulting domain more easily?

Elaboration tolerance [2]: How do we ensure that domains can initially be modeled at a high level of abstraction, 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 article, we investigate the applicability of the object-oriented paradigm [3,4] to answering these questions. 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 the attributes and methods of the *superclass* and possibly adding its own attributes and methods or redefining some of the 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, as does the ability to extend existing classes with new functionality in a structured and well-defined manner and to override existing functionality by re-implementing inherited methods. The modularity and reusability of a model are improved by modeling self-contained classes that are independent of the implementations of other classes.

The object-oriented concepts used in this article could potentially be applied to many different logics for reasoning and change, as long as they provide a certain minimum amount of expressivity. However, a proper demonstration of the viability of the approach requires a varied set of concrete examples. For these examples we have chosen to use a single logic: TAL-C [5].

In the first part of the article, we will introduce TAL-C (Section 2), show how domains can be modeled in TAL-C in an object-oriented manner (Sections 3 and 4)

and discuss some more complex issues related to object-orientation (Section 5) and how this affects elaboration tolerance (Section 6). Then, the ideas covered in the first part will be applied to the Missionaries and Cannibals domain (Section 7). The 19 elaborations of this domain defined by McCarthy in his paper on elaboration tolerance [2] will also be covered (Section 8), and a way of actually solving the problems within the logic is discussed (Section 9). An object-oriented model of the Traffic World domain [6] is briefly mentioned (Section 10). Finally, we conclude with related work (Section 11) and a discussion of the results (Section 12).

2 The TAL family and the TAL-C Logic

TAL, Temporal Action Logics [7], is a family of non-monotonic temporal logics with discrete linear time originating from the Features and Fluents framework [8] and developed for reasoning about action and change in dynamic and incompletely specified domains.

The TAL family contains a number of logics incrementally developed to provide robust solutions to a number of problems in the area of reasoning about action and change. Current members of the TAL family allow the modeling of actions with duration, context-dependent actions, incompletely specified timing of actions, and non-deterministic actions. They also provide a robust solution to the frame problem [9]. Actions can have side effects, and chains of side effects are handled correctly, providing one approach to solving the ramification problem [10]. The TAL-Q logic provides one approach towards solving the qualification problem [11]. Recent work also includes support for delayed effects of actions [12] and, in TAL-C, concurrent actions [5]. All of these features have a corresponding formal semantics, and TAL-C is also used as the semantic basis for TALplanner [13,14].

The TAL logics are narrative-based, and use two languages for representing and reasoning about narratives. The surface language $\mathcal{L}(\mathrm{ND})$ (Narrative Description Language, described in more detail below) provides a convenient high-level macro notation for describing narratives, and can be extended in various ways in different logics in the TAL family. A narrative in any version of $\mathcal{L}(\mathrm{ND})$ can be mechanically translated into a common logical base language $\mathcal{L}(\mathrm{FL})$, where the frame, ramification and qualification problems are handled using a form of circumscription [15] called filtered circumscription [16]. Due to constraints on the structure of an $\mathcal{L}(\mathrm{ND})$ narrative, the second-order theory resulting from applying circumscription can always be translated into a logically equivalent first-order theory, which is then used to reason about the narrative. The formal details are presented in [7] as well as in Appendices A and B.

This article will use TAL-C as a basis for applying concepts from object-oriented modeling. A subset of this logic is implemented in the research tool VITAL [17], a

platform-independent Java tool that can be downloaded from the WWW. All narratives belonging to the subset supported by VITAL have a finite number of models, and VITAL uses constraint propagation techniques to generate all models (or any given number of models) of such narratives. This provides us with an experimental platform where object-oriented narratives can be tested.

In the remainder of this section, we will use a concrete narrative example to provide an intuitive understanding of TAL-C. This will provide a basis for the object-oriented extensions presented in Section 3.

2.1 TAL-C Narrative Descriptions

A narrative description in the TAL surface language $\mathcal{L}(\mathrm{ND})$ consists of two parts: The *narrative background specification* (NBS) and the *narrative specification* (NS).

The narrative background specification contains generic information about the domain that is being modeled. This includes a narrative type specification, containing type descriptions for the features [1] , value domains, and actions that are present in the domain. It also includes a set of labeled narrative statements containing action definitions (action schemas, labeled acs), domain constraints representing static constraints that are always satisfied in the domain (acc, also called acausal constraints), dependency constraints representing directional or causal dependencies between fluents (dep), and persistence properties of fluents (per).

The narrative specification contains information specific to a particular reasoning problem within a problem domain, and includes observations of actual fluent values in the initial state or any other state (observation statements, labeled obs) and information about which actions were performed, with which arguments, and when (action occurrence statements, labeled occ).

Since narrative examples used in the literature have traditionally been quite simple, the narrative type specification has usually either been considered to be implicit in the remainder of the narrative specification or been described in the main text of the article. In this article, we will instead use the input syntax for VITAL for the narrative type specification.

[1] A feature is similar to a state variable. When viewed as a function of time, it is called a fluent.

We will now provide a concrete TAL-C narrative example using a variation of the well-known hiding turkey scenario. This requires the following narrative type specification:

domain boolean :elements { true, false }
feature alive, deaf, hiding, loaded :domain boolean
action Load, Fire

The following statements comprise the remainder of the narrative type specification. Explanations will be provided below.

per_1 $\forall t.t > 0 \rightarrow \mathit{Per}(t, \mathsf{alive}) \wedge \mathit{Per}(t, \mathsf{deaf}) \wedge \mathit{Per}(t, \mathsf{hiding}) \wedge \mathit{Per}(t, \mathsf{loaded})$
per_2 $\forall t.\mathit{Dur}(t, \mathsf{noise}, \mathsf{false})$
dep_1 $\forall t.[t]\ \neg\mathsf{hiding} \wedge \neg\mathsf{deaf} \wedge \mathsf{noise} \rightarrow \mathbf{Set}([t+1]\ \mathsf{hiding})$
dep_2 $\forall t.[t, t+9]\ \mathsf{hiding} \wedge \neg\mathsf{noise} \rightarrow \mathbf{Set}([t+10]\ \neg\mathsf{hiding})$
acs_1 $[t_1, t_2]\ \mathsf{Load} \rightsquigarrow \mathbf{Set}([t_2]\ \mathsf{loaded}) \wedge \mathbf{Set}((t_1, t_2]\ \mathsf{noise})$
acs_2 $[t_1, t_2]\ \mathsf{Fire} \rightsquigarrow ([t_1]\ \mathsf{loaded} \wedge \neg\mathsf{hiding} \rightarrow \mathbf{Set}(t_2]\ \neg\mathsf{alive})) \wedge$
$([t_1]\ \mathsf{loaded} \rightarrow \mathbf{Set}([t_2]\ \neg\mathsf{loaded}))$

Finally, the following is the narrative specification, which specifies what happens in this particular scenario within the hiding turkey domain:

obs_1 $[0]\ \mathsf{alive} \mathrel{\hat{=}} \mathsf{true} \wedge \mathsf{hiding} \mathrel{\hat{=}} \mathsf{true} \wedge \mathsf{loaded} \mathrel{\hat{=}} \mathsf{false}$
occ_1 $[1, 4]\ \mathsf{Load}$
occ_2 $[5, 6]\ \mathsf{Fire}$

In this variation of the hiding turkey scenario, there is a turkey which we observe being alive and not hiding in the initial state at time 0 (specified in the observation statement obs_1). The turkey may or may not be deaf – since there is no mention of this fluent, it is not constrained to be either true or false. The observation statement uses the notation $[\tau]\ \phi$, which means that the fluent formula ϕ holds at time τ. The fluent formula $f \mathrel{\hat{=}} v$ denotes the fact that the fluent f takes on the value v at the given timepoint. For boolean fluents the shorthand notation $f \stackrel{\text{def}}{=} f \mathrel{\hat{=}} \mathsf{true}$ and $\neg f \stackrel{\text{def}}{=} f \mathrel{\hat{=}} \mathsf{false}$ is also allowed.

The scenario also involves a gun, which is not loaded in the initial state (also specified in obs_1).

Most of the fluents are *persistent* (per_1), meaning that their values persist to the next timepoint unless explicitly assigned a new value using the **Set** macro (previously called I in most papers, and formally defined in Appendix A). However, noise is a *durational* fluent with a default value false (per_2). It can only be true at those timepoints where it is explicitly assigned the value true, and at all other timepoints it automatically reverts to being false. This reflects the common-sense notion that

Accepted for publication in Artificial Intelligence Journal, 2003

there is no noise unless someone or something is currently making noise. If a fluent is not declared to be persistent or durational, it is *dynamic*. Since no persistence or default value assumption is applied, the fluent can vary freely over time to satisfy observations and domain constraints.

An interesting property of the turkey is that it is afraid of sounds. If it is not deaf and there is some noise, it will immediately hide. This fact cannot be modeled as a domain constraint, since such constraints cannot provide a sufficient cause for the noise fluent to change values. Instead, it is modeled using the dependency constraint dep_1, which explicitly assigns hiding the value true using the **Set** macro. When there has been no noise for ten consecutive timepoints, it will finally stop hiding, which is modeled using another dependency constraint (dep_2) – note that intervals $[\tau, \tau']$ are allowed.

There are two actions at our disposal. We can Load the gun (acs_1), which ensures that the gun is loaded when the action has been executed but also makes some noise throughout the duration of the action: The action definition forces noise to be true in the entire interval $(t_1, t_2]$, and thereafter noise will automatically revert to its default value, false. We can also Fire the gun (acs_2), which results in the gun no longer being loaded – and if the gun was loaded when the Fire action was invoked, and the turkey was not hiding, the turkey will die.

In fact, in this particular scenario, we do Load the gun between 1 and 4 (occ_1), and we Fire it between 5 and 6 (occ_2). If the turkey is deaf, it will not hide and ends up being shot. Otherwise, it hears the noise, hides, and emerges from hiding ten timepoints later. Since it was not specified whether the turkey was deaf or not, there will be two classes of model for this scenario: One where the turkey dies and one where it remains alive.

Apart from being more complex than many traditional benchmark problems in the RAC community (the even more well-known Stanford Murder Mystery requires only four short statements), this narrative is fairly representative of the area. The nine statements are ordered by type, with no special regard to the structure of the problem. The fluents are also unstructured in the sense that there is no indication that alive and hiding refer to properties of a turkey while loaded and noise do not.

Although the hiding turkey domain is still comprehensible in this unorganized form, it is clear that some additional structure will be valuable when modeling more complex domains. The following section presents a way of applying the object-oriented paradigm to modeling such domains.

2.3 A Note on Fluents, Sorts and Types

Although this is not apparent from the turkey domain example presented above, TAL-C is order-sorted and allows the use of a hierarchy of arbitrary finite value domains. Fluents take on values from a specific sort (possibly the standard sort $\mathsf{boolean} = \{\mathsf{true}, \mathsf{false}\}$), and both fluents and actions can take arguments of specific types.

For example, it would be possible to define a value domain gun containing the three guns gun1, gun2 and gun3. The narrative could then be extended to use the boolean fluent $\mathsf{loaded}(\mathsf{gun})$ together with the two actions $\mathsf{Load}(\mathsf{gun})$ and $\mathsf{Fire}(\mathsf{gun})$ taking a gun as an argument.

Variables are typed and range over the values belonging to a specific sort. Although the sort is sometimes specified explicitly, it is more common to simply give the variable the same name as the sort but (like all variables) written in italics, possibly with a prime and/or an index. For example, the variables gun, gun' and gun_3 would be of the sort gun. Similarly, variables named t or τ are normally temporal variables, and variables named n are normally integer-valued variables.

3 Basic Object-Oriented Modeling in TAL-C

As has been shown previously [5,12,18], the TAL logics are flexible and fine-grained logics 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 $\mathcal{L}(\mathrm{ND})$ syntax will be introduced. These extensions are not essential, since the new macros and statement classes can mechanically be translated into the older syntax. The translations are implemented in the research tool VITAL.

The remainder of this section will show how classes are declared and how to instantiate objects of a specific class. We will then go on to discuss how to declare and use attributes (fields), and how to use three different types of methods: Accessors, mutators, and constraint methods. This provides the basic functionality for the object-oriented modeling of complex domains in TAL-C. Section 4 will cover additional topics such as how to override a method.

3.1 Defining Classes and Objects

In TAL, domains are traditionally modeled using an unstructured set of boolean or non-boolean fluents, each of which can take a number of arguments belonging to specific value domains.

In our 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.

Given the approach being used, it would be easy to introduce a class alias for the ordinary domain declaration statement. However, this would mean that any class declaration statement would have to explicitly enumerate all objects belonging to the class. Instead, a new, more flexible syntax is introduced which allows class and object declarations to be separated.

3.1.1 Defining Classes

The narrative type specification syntax in VITAL is extended to allow two forms of class declaration statement. A statement on the form class NEWCLASS declares a new *top-level class* named NEWCLASS, without a parent. Usually this is only used for the OBJECT class. A statement on the form class SUB *extends* SUPER declares a new subclass named SUB, with the parent class (superclass) SUPER. This makes SUB a *direct subclass* of SUPER, and SUPER is a *direct superclass* of SUB.

A class SUB is a *subclass* of SUPER iff it is a direct subclass of SUPER there is an intermediate class INTER such that SUB is a direct subclass of INTER and INTER is a subclass of SUPER. The *superclass* concept is defined similarly.

A simple water tank domain will be used as a running example. This domain requires the standard root class OBJECT together with a domain TANK for water tanks. We are also interested in modeling a special type of tank, a FLOWTANK, which may have a flow of water into or out of the tank, as well as PIPEs between the tanks.

class OBJECT
class TANK *extends* OBJECT
class FLOWTANK *extends* TANK
class PIPE *extends* OBJECT

3.1.2 Defining Objects

Objects are declared in the narrative type specification using object statements (labeled obj). Declaring an object as a member of a class naturally also makes it a member of its superclasses: Any FLOWTANK is automatically also a TANK and an OBJECT.

obj **tank1** : TANK
obj **tank2**, **tank3** : FLOWTANK
obj **pipe1** : PIPE

Note that since classes correspond to value domains, it is possible to quantify over all objects belonging to a given class. Also note that objects are not created at any particular timepoint. They are declared in the narrative specification and exist at all timepoints.

3.1.3 Translation

Class declarations and object declarations cannot be translated in isolation. Instead, the complete set of class and object declarations are translated into TAL-C in the following manner.

An object o is explicitly declared to belong to the class CL iff there is an object declaration statement on the form obj $\ldots$, o_i, $\ldots$: CL. An object o is declared to belong to the class CL iff it is explicitly declared to be belong to CL or to one of the superclasses of CL.

Each class declaration statement class NEWCLASS for a top-level class NEWCLASS is translated into the domain declaration statement domain NEWCLASS :elements $\{$ $o_1, \ldots, o_n$ $\}$, where the objects $o_1, \ldots, o_n$ are exactly those objects that are declared to belong to NEWCLASS.

Each class declaration statement class SUB *extends* SUPER for a non-top-level class SUB is translated into the domain declaration statement domain SUB :extends SUPER :elements $\{$ $o_1, \ldots, o_n$, where the objects $o_1, \ldots, o_n$ are exactly those objects that are declared to belong to SUB.

This leads to the following VITAL domain definitions for the classes and objects declared above:

domain OBJECT :elements { **pipe1**, **tank1**, **tank2**, **tank3** }
domain TANK :elements { **tank1**, **tank2**, **tank3** }
domain FLOWTANK :elements { **tank2**, **tank3** }
domain PIPE :elements { **pipe1** }

3.2 Using Attributes

As usual in object-oriented languages, each object can be associated with a set of attributes, also known as fields. All objects of a certain class share the same attributes, but the specific values of the attributes may differ between the objects. Below, we show how attributes are modeled in TAL-C, how they are initialized, and how they can be changed at specific points in time.

3.2.1 Defining Attributes

All attributes are specified in attribute declarations (labeled attr). For example, any TANK has a current volume, a maximum volume, and a base area, all of which are `Real` values.[2] These attributes are persistent: They will not change unless explicitly changed. This is specified as follows:

attr TANK.volume : `Real`
attr TANK.maxvol : `Real`
attr TANK.area : `Real`

It is also possible to define attributes with arguments, which provides functionality similar to the use of arrays or mappings in programming languages. For example, if any water tank must keep track of exactly which pipes it is connected to, this can be modeled using a boolean attribute connected taking a pipe as an argument:

attr TANK.connected(PIPE) : `boolean`

An attribute is automatically translated into a feature taking one additional argument – an object of the class to which the attribute belongs. Thus, the declarations above are translated into the four TAL fluents volume(TANK) : `Real`, maxvol (TANK) : `Real`, area(TANK) : `Real`, and connected(TANK, PIPE) : `boolean`. Since time-dependent fluents are used, any attribute can vary over time in a natural manner.

[2] Since TAL currently requires finite domains, it is necessary to specify upper and lower bounds on the `Real` domain as well as the desired precision. This is also true for the `Integer` domain which will be used in later examples. However, these limitations are not relevant to the modeling issues covered in this article.

More formally, an attribute declaration attr CLS.attr$(s_1, \ldots, s_n) : s$ where $n \geq 0$ is translated into a feature declaration feature attr$(\text{CLS}, s_1, \ldots, s_n) : s$.

Using standard TAL-C syntax, the volume attribute of **tank1** is denoted by volume (**tank1**). To permit the use of the standard object-oriented syntax **tank1**.volume, we define *obj*.attr$(x_1, \ldots, x_n) \stackrel{\text{def}}{=}$ attr$(obj, x_1, \ldots, x_n)$, where $n \geq 0$; if $n = 0$, the parentheses may be omitted. This syntax will also be applied to method invocations.

3.2.2 Attributes in Subclasses

Due to the use of the order-sorted type structure in TAL-C, subclasses automatically inherit the attributes of their parents, as in ordinary object-oriented languages. For example, **tank1**, **tank2** and **tank3** all have a volume, despite that the latter two are declared as FLOWTANK objects.

Naturally, subclasses can also add new attributes. For example, the FLOWTANK class keeps track of the current flow of water in or out of the tank, which is modeled as a flow attribute:

attr FLOWTANK.flow : `Real`

3.2.3 Initializing Attributes

Although it would have been possible to introduce special syntax for object initialization, similar to constructors in standard object-oriented languages, this only appears to be natural in the case where complete information about all objects is available.

The TAL logics allow the use of incomplete information – for example, due to sensor accuracy, one might only know that the initial volume of water in a tank is less than 0.02. Therefore, we still use plain TAL-C observation statements to partially or completely initialize attributes at time 0.

obs $\forall tank.[0]$ *tank*.volume ≤ 0.02
obs $[0]$ **tank2**.flow $\hat{=} 0 \wedge$ **tank3**.flow $\hat{=} 0.12$

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: Accessors (which query the state of an object), mutators (which are called in order to change the state of an object), and constraint methods (which are not explicitly invoked but are active at all timepoints).

3.3.1 Accessors

Accessors are used for querying the state of an object. This can be done simply by retrieving the current value of an attribute, or by performing arbitrarily complex calculations as long as these calculations can be expressed within the logic being used.

An accessor is modeled using a *return value fluent*, a dynamic (non-persistent, non-durational) fluent that takes on the desired return value at all timepoints. For example, a simple query_volume() method for a water tank can be modeled by introducing a dynamic fluent query_volume(TANK) : Real and adding the following domain constraint:

acc $[t]$ *tank*.query_volume() $\hat{=}$ *value*$(t,$ *tank*.volume$)$

Although this type of accessor may not appear very useful at first glance, the intention is that the attributes of a class (such as volume in TANK) should be considered private within that class, and that external callers should only use the externally available interface, such as the query_volume accessor. Actually enforcing this intention would require additional support from the tools being used to reason about an object-oriented narrative.

A slightly more complex accessor might determine whether the tank is full, which is the case if its volume equals its maximum volume (maxvol). This is done by declaring a dynamic return value fluent query_full(TANK) : boolean and using the following domain constraint:

acc $[t]$ *tank*.query_full() $\leftrightarrow$ *value*$(t,$ *tank*.volume$)$ $=$ *value*$(t,$ *tank*.maxvol$)$

3.3.2 Mutators

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, it is first necessary to define a boolean durational invocation fluent method $(\text{CLASS}, s_1, \ldots, s_n)$ with default value false. The method implementation consists of a dependency constraint that is triggered for an object *obj* only when *obj*.method $(x_1, \ldots, x_n)$ is true. For example, a mutator set_volume(Real) can be defined in class TANK as follows:

Accepted for publication in Artificial Intelligence Journal, 2003

$$\mathsf{per}\ \forall t, \mathit{tank}, v \in \texttt{Real}.\mathit{Dur}(t, \mathit{tank}.\mathsf{set\_volume}(v), \mathsf{false})$$
$$\mathsf{dep}\ \forall t, \mathit{tank}, v \in \texttt{Real}.[t]\ \mathit{tank}.\mathsf{set\_volume}(v) \rightarrow \mathbf{Set}([t]\ \mathit{tank}.\mathsf{volume} \mathrel{\hat{=}} v)$$

Calling the method requires making the invocation fluent true for the appropriate arguments at the timepoint when the method should be invoked. As usual, this is done using the **Set** macro, and therefore a TAL dependency constraint is required. For example, the volume of **tank1** can be set to 4.5 at time 2 as follows:

$$\mathsf{dep}\ \mathbf{Set}([2]\ \mathbf{tank1}.\mathsf{set\_volume}(4.5) \mathrel{\hat{=}} \mathsf{true})$$

This is simplified further by defining $\mathbf{Call}(\tau, f) \stackrel{\mathsf{def}}{=} \mathbf{Set}([\tau]f \mathrel{\hat{=}} \mathsf{true})$:

$$\mathsf{dep}\ \mathbf{Call}(2, \mathbf{tank1}.\mathsf{set\_volume}(4.5))$$

3.3.3 Constraint Methods

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:

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

This concludes the discussion of the most basic concepts in object-orientation: Classes, objects, attributes, and methods. The following section will show how to reify the class structure in order to model method overriding in TAL-C, while Section 5 demonstrates how some additional object-oriented concepts, such as abstract classes and final methods, can be modeled.

4 Inheritance and Overriding

Although the concepts presented in the previous section are sufficient for modeling many domains, it is still possible to improve the elaboration tolerance of the models considerably by introducing the object-oriented concept of *overriding*: Allowing a subclass to re-implement a method in order to refine or specialize it.

This requires a way of disabling a method implementation that is inherited from a superclass, which is facilitated by providing the logic formulas with some additional information about the class structure used in a domain model.

Since the TAL logics have no built-in support for allowing logic formulas to inspect the class (sort) structure of a particular domain, it is necessary to reify this structure. This can of course be done mechanically, and support for this is built into current versions of VITAL [17].

The class structure is reified by mechanically constructing a TAL value domain `classname` containing all class names, and declaring and initializing a persistent boolean fluent [3] 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:

domain `classname` :elements { OBJECT, TANK, FLOWTANK }
feature subclass(`classname`, `classname`) :domain boolean
per $\forall t, classname_1, classname_2 . t > 0 \rightarrow$
Per$(t,$ subclass$(classname_1, classname_2))$

obs $\forall c_1 \in$ `classname`$, c_2 \in$ `classname`
$[0]$ subclass$(c_1, c_2) \leftrightarrow$
$((c_1 =$ FLOWTANK $\wedge\ c_2 =$ OBJECT$) \vee$
$(c_1 =$ FLOWTANK $\wedge\ c_2 =$ TANK$) \vee$
$(c_1 =$ TANK $\wedge\ c_2 =$ OBJECT$))$

Note that since subclass is persistent, it is sufficient to provide a value at time 0. This value will automatically propagate to all timepoints.

In addition to this, it is sometimes necessary to be able to identify the exact type of a certain object. To this end, an attribute class of type `classname` is added to the root class OBJECT:

attr OBJECT.class : `classname`

This attribute is also initialized automatically during the translation of the object declaration statements. In the water tank example, the following observations would be generated:

obs $[0]$ **tank1**.class $\hat{=}$ TANK
obs $[0]$ **tank2**.class $\hat{=}$ FLOWTANK
obs $[0]$ **tank3**.class $\hat{=}$ FLOWTANK
obs $[0]$ **pipe**.class $\hat{=}$ PIPE

It should be emphasized that these domains and fluents are created automatically during the translation process and need not be explicitly defined by the user.

[3] Although we do not intend to change subclass relations over time, TAL-C has no support for time-independent functions and therefore a fluent must be used.

Suppose that a method method is defined and implemented in a class CLASS $\in$ classname. This implementation of method will be active for any object of type CLASS, including objects belonging to subclasses of CLASS.

When a new subclass SUB is created, we may want to override some of the methods defined in the superclass CLASS. This means not only adding a new implementation of the method for objects in SUB, but also *disabling* the old implementation for those objects.

To allow this to be modeled in TAL-C it is necessary to reify the concept of overriding a method. We introduce the boolean fluent override(SUB, method, CLASS) expressing the fact that for objects belonging to SUB, any implementation of method in the superclass CLASS is overridden and should be disabled. This fluent is durational with default value false, since overriding should only occur when explicitly forced.

All method implementations should then be conditionalized on not being overridden, and should explicitly override implementations in superclasses.

The former is achieved by adding a suitable override expression in the precondition of each method. For example, when set_volume mutator declared in class TANK is called for an object *tank*, the exact type of that object is *tank*.class (which may be TANK or FLOWTANK). Thus, the method should be disabled if for objects of this type (*tank*.class), the implementation of set_volume in the class TANK is overridden – in other words, if override(*tank*.class, set_volume, TANK). The method is therefore conditionalized as follows:

dep $\forall t, \mathit{tank} \in \text{TANK}, f \in \texttt{Real}$
$[t]\ \mathit{tank}.\text{set\_volume}(f) \wedge \neg\text{override}(\mathit{tank}.\text{class}, \text{set\_volume}, \text{TANK}) \rightarrow$
$\mathbf{Set}([t]\ \mathit{tank}.\text{volume} \mathrel{\hat{=}} f)$

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

dep $\forall t, \text{SUPER} \in \texttt{classnames}, \text{SUB} \in \texttt{classnames}$
$([t]\ \text{subclass}(\text{CURRENTCLASS}, \text{SUPER})) \wedge$
$([t]\ \text{subclass}(\text{SUB}, \text{CURRENTCLASS}) \vee \text{SUB} = \text{CURRENTCLASS}) \rightarrow$
$\mathbf{Set}([t]\ \text{override}(\text{SUB}, \text{method}, \text{SUPER}))$

This states that when a method is re-implemented in CURRENTCLASS, its inherited implementation from any superclass SUPER is disabled for any object whose type SUB is either exactly CURRENTCLASS or a subclass of CURRENTCLASS.

Accepted for publication in Artificial Intelligence Journal, 2003

For convenience, the macro **DisableInherited**(CURRENTCLASS, method) will be used as a shorthand for statements of this type. This yields the following final definition of the set_volume mutator:

dep **DisableInherited**(TANK, set_volume)
dep $\forall t, \mathit{tank} \in \text{TANK}, v \in \texttt{Real}$
$[t]\ \mathit{tank}.\text{set\_volume}(v) \wedge \neg\text{override}(\mathit{tank}.\text{class}, \text{set\_volume}, \text{TANK}) \rightarrow$
Set$([t]\ \mathit{tank}.\text{volume} \mathrel{\hat{=}} v)$

5 Additional Object-Oriented Concepts

This section will briefly present some additional ideas regarding the use of object-oriented modeling in a logic for reasoning about action and change. These ideas build on the basic concepts presented in the previous two sections, but will not be developed in the same level of detail. Rather, they are intended to demonstrate the flexibility of the paradigm and show how it could be extended and modified in various directions depending on the needs of the user.

5.1 Multiple Method Implementations

In the examples presented previously, a method always has a single implementation. However, there is no reason why this always has to be the case. For example, a mutator could consist of multiple dependency constraints, all of which are triggered by the same invocation fluent. This allows a more modular implementation of complex methods. It also permits a subclass to *add* to the implementation of a method, rather than replace it, simply by not calling the **DisableInherited**(CLASS, method) macro to disable the implementation provided by the superclass. This resembles the ability to call a superclass implementation of a method using super.method(...) in the Java programming language.

5.2 Preventing Overriding: Final Methods

In some object-oriented programming languages, a method implementation can be marked as "final", meaning that it cannot be overridden in a subclass.

Final methods can be defined in TAL-C by stating that they are never overridden. For example, the set_volume method from Section 4.2 could be made final by adding the following statement:

acc $\forall t, \mathit{tank} \in \text{TANK}.[t]\ \neg\text{override}(\mathit{tank}.\text{class}, \text{set\_volume}, \text{TANK})$

Accepted for publication in Artificial Intelligence Journal, 2003

Unlike most programming languages, this form of type checking is dynamic rather than static. If a method is overridden despite being final, this will generate an inconsistent narrative rather than an error during translation. VITAL will detect such inconsistencies and report the error to the user.

5.3 Forcing Overriding: Abstract Methods

While final methods are implemented and cannot be overridden in subclasses, abstract methods are *not* implemented and *must* be overridden in all subclasses. The following statement can be used to declare that the get_color method is abstract in the class TANK:

acc $\forall t, \mathit{tank} \in \text{TANK}.[t]$ override$(\mathit{tank}.\text{class}, \text{get_color}, \text{TANK})$

Note that this statement in itself is not sufficient for permitting the override fluent to be true. The fluent is durational, and can only take on the value true if it is explicitly *assigned* that value, which is not the case in this formula. Instead, the formula states that someone else must have assigned it the value true using the **Set** macro, which would be done indirectly by an overriding method declaration using the **DisableInherited** macro.

5.4 Abstract Classes

An abstract class cannot be instantiated. Such a class can be modeled using a simple constraint on the following form:

acc $\forall t \neg \exists \mathit{object}.[t]$ $\mathit{object}.\text{class} \mathrel{\hat{=}} \text{CLASS}$

5.5 Class Methods

All methods shown up to now have been instance methods. For example, set_volume is called for an instance of the TANK class, and only alters the volume of that specific instance. It is also possible to model class methods, which are associated with the class itself rather than with an instance.

A class accessor method can be defined in TAL-C using a return value fluent that does not take an object as its first argument. Similarly, a class mutator can be defined using an invocation fluent that does not take an object as its first argument. For example, all tank volumes can be reset to zero using the following class method in the TANK class:

dep $\forall t.[t]$ set_zero_volume$() \wedge \neg$override$(\text{TANK}, \text{set_zero_volume}, \text{TANK}) \rightarrow$
$\forall tank.$**Set**$([t]\ tank.\text{volume} \hat{=} 0.0)$

Note that this method is called directly, as in **Call**$(7, \text{set\_zero\_volume}())$, without specifying a tank object as in **Call**$(7, \textbf{tank1}.\text{set\_volume}(0))$.

5.6 Access Control

For mutators, a form of cooperative access control can be implemented by adding to the invocation fluent another argument representing the caller. Using the set_volume mutator as an example, the following changes would be made:

dep **DisableInherited**$(\text{TANK}, \text{set\_volume})$
dep $\forall t, tank \in \text{TANK}, caller \in \text{TANK}, v \in \texttt{Real}$
$[t]\ tank.\text{set\_volume}(caller, v) \wedge \neg\text{override}(tank.\text{class}, \text{set\_volume}, \text{TANK}) \rightarrow$
Set$([t]\ tank.\text{volume} \hat{=} v)$

In this definition, the caller argument must be a TANK, and consequently only a TANK can call the set_volume method. This is similar to a protected method in Java, and could possibly be used to help ensure that encapsulation is respected. However, this only provides a purely cooperative form of access control, since anyone wanting to call set_volume() could in principle simply send an arbitrary tank object as the caller.

6 Elaboration Tolerance through Object-Orientation

According to McCarthy [2], 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.

The structuring of objects, fluents, domain constraints and dependency constraints into a well-defined set of named classes, attributes and methods is a powerful tool for increasing the readability of a domain definition. This helps provide a better understanding of the domain, which is in itself a very important prerequisite for being able to adapt and extend the definition.

The use of inheritance makes it possible to specialize a class, adding new attributes, methods and constraints while reusing those features from the superclass that are still useful in the new subclass. 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).

While the creation of a subclass does not alter the behavior of its superclass, it is also possible to add new attributes and methods directly to an existing class without the need to modify the existing parts of the class definition.

Adding a new class requires changes to the `classname` domain and the subclass fluent. These changes are done automatically at translation time. Adding new methods may also yield a new definition of the automatically generated *Occlude* predicate (the TAL approach to solving the frame problem, as described in Appendices A and B). However, the new definition can be created by analyzing the new methods in isolation and adding new disjuncts to the existing definition of *Occlude*. It is not necessary to start over from the beginning because a new class is added or because a method is overridden.

The elaboration tolerance of this approach will now be tested using a concrete example domain.

7 Missionaries and Cannibals

McCarthy [2] illustrates his ideas regarding elaboration tolerance with 19 elaborations of the Missionaries and Cannibals Problem (MCP). We will begin by modeling the basic, unelaborated domain using the object-oriented constructions presented above. In the next section we will show that the ability to override methods and to add new methods and attributes in subclasses provides a natural way to model many of the elaborations. Section 9 shows how the problem instances can be solved by generating plans within the logic.

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

We will define classes for objects, boats, places, and banks (Figure 1). Like Lifschitz [19], we will initially model missionaries and cannibals as *groups* of a certain size rather than as individuals, despite the fact that a few of the elaborations do require individuals to be treated as such; this is also done to provide a better test for elaboration tolerance. 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.

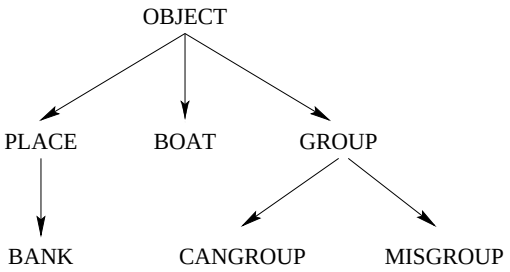

Fig. 1. Classes in the Missionaries and Cannibals Domain

7.2 Object

The root class OBJECT has a pos attribute representing its position, which is a PLACE (Section 7.3):

```
class OBJECT
  attr OBJECT.pos : PLACE
```

The following methods are available for accessing and changing the position:

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

dep **DisableInherited**(OBJECT, query_pos)
dep $[t]$ ¬override(*object*.class, query_pos, OBJECT) $\rightarrow$
  Set($[t]$ *object*.query_pos() $\hat{=}$ *value*(t, *object*.pos)) □

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

dep **DisableInherited**(OBJECT, set_pos)
dep $[t]$ ¬override(*object*.class, set_pos, OBJECT) $\wedge$ *object*.set_pos(*place*) $\rightarrow$
  Set($[t]$ *object*.pos $\hat{=}$ *place*) □

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

7.3 Place

The standard problem contains three different places: The left and right river bank and onboard the boat. This is modeled as a generic class PLACE with a subclass BANK.

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

class PLACE *extends* OBJECT
  attr PLACE.connection(PLACE) : `boolean`

Since the PLACE onboard the boat will be connected to the bank where it is currently located, and since the boat will move between the two banks, the connection attribute will change dynamically over time. Therefore two mutator methods are available, in addition to the standard query method.

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

dep **DisableInherited**(PLACE, query_connection)
dep $[t]$ ¬override($place$.class, query_connection, PLACE) $\rightarrow$
  Set($[t]$ $place$.query_connection($place'$) $\hat{=}$ $value(t, place.\text{connection}(place'))$)
  □

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

dep **DisableInherited**(PLACE, add_connection)
dep $[t]$ ¬override($place$.class, add_connection, OBJECT) $\wedge$
    $place$.add_connection($place'$) $\rightarrow$
  Set($[t]$ $place$.connection($place'$) $\hat{=}$ true) $\wedge$
  Set($[t]$ $place'$.connection($place$) $\hat{=}$ true) □

Mutator remove_connection(PLACE): Removes the connection from this PLACE to another PLACE.

Accepted for publication in Artificial Intelligence Journal, 2003

dep **DisableInherited**(PLACE, remove_connection)
dep $[t]$ ¬override($place$.class, remove_connection, OBJECT) ∧
$place$.remove_connection($place'$) →
Set($[t]$ $place$.connection($place'$) $\hat{=}$ false) ∧
Set($[t]$ $place'$.connection($place$) $\hat{=}$ false) □

7.4 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.

class BANK *extends* PLACE

This class adds no new methods or attributes. Instead, the constraints on a BOAT will guarantee that it is always located at a BANK.

7.5 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.

class GROUP *extends* OBJECT
attr GROUP.size : `Integer`

Accessor query_can_move_to(GROUP): In the basic domain, people can move from one group to another only if they are groups of the same type and the two groups are connected. For example, people cannot move from a missionary group to a cannibal group, or teleport from the left bank to the right bank. For simplicity, we make the return value fluent durational with default value false, and explicitly set it to true only when necessary.

dep **DisableInherited**(GROUP, query_can_move_to)
dep $[t]$ ¬override($group$.class, query_can_move_to, GROUP) ∧
$group$.query_pos().query_connection($group'$.query_pos()) ∧
$group$.class $\hat{=}$ $group'$.class) →
Set($[t]group$.query_can_move_to($group'$) $\hat{=}$ true) □

Accepted for publication in Artificial Intelligence Journal, 2003

Mutator modify_group(GROUP, n): Calling $group$.modify_group$(group_2, n)$ moves n people from *group* to $group_2$, if n is positive – otherwise, it moves $|n|$ people in the other direction. It is the caller's responsibility to use query_can_move_to() to ensure that the change is in fact "legal", and to ensure that a sufficient number of people is available in the source group. It is also the caller's responsibility to ensure that symmetry is retained: If $group$.modify_group$(group_2, n)$ is called, $group_2$.modify_group $(group, -n)$ must also be called.

The implementation of this method is somewhat complex due to the fact that people could move concurrently between multiple groups. For example, one person could move from **group1** to **group2** while another moves from **group2** to **group3** and two from **group3** to **group1**. The cumulative effects of these concurrent method calls must be taken into account.

For this reason, modify_group does not follow the standard pattern where each invocation triggers a separate instance of a formula. Instead, a single dependency constraint sums the arguments of all concurrent invocations: [4]

dep **DisableInherited**(GROUP, modify_group)
dep $[t]$ ¬override($group$.class, modify_group, GROUP) →
Set($[t+1]$ $group$.size $\hat{=}$ $value(t, group.\text{size}) + \sum_{\{\langle g,x\rangle \mid g \in \text{GROUP} \wedge [t]\ group.\text{modify\_group}(g,x)\}} x$

□

The macro **people_at**$(\tau, \text{GROUP}, place)$ will denote the number of people at *place* of the given type GROUP at time τ:

$$\textbf{people\_at}(\tau, \text{GROUP}, place) = \sum_{\{g \mid g \in \text{GROUP} \wedge [\tau]\ g.\text{query\_pos}() \hat{=} place\}} value(\tau, g.\text{query\_size}())$$

For example, given that **left** denotes the left bank, the macro expression **people_at** $(7, \text{CANGROUP}, \textbf{left})$ denotes the number of cannibals on the left bank at time 7, and **people_at**$(7, \text{GROUP}, \textbf{left})$ denotes the total number of people on the left bank at time 7.

7.6 Cannibals

A CANGROUP is a group of cannibals. The class extends GROUP and adds one new method.

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

class CANGROUP *extends* GROUP

Constraint eat_missionaries(): Specifies that there cannot be more cannibals than missionaries at any place. This constraint rules out any state where the cannibals would be able to eat a missionary.

Note that whenever a boat is at a river bank, anyone in the boat is considered to be at the same place as anyone on the bank. For this reason we define the macro **people_in_boats_near**$(\tau, \text{GROUP}, \mathit{place})$, denoting the number of people in boats at the given place *place*, belonging to a group of the given type GROUP, at time τ:

$$\textbf{people\_in\_boats\_near}(\tau, \text{GROUP}, \mathit{place}) \quad = \sum_{\{\langle \mathit{boat}, g\rangle \mid [\tau]\ g.\mathsf{query\_pos}() \hat{=} \mathit{boat}.\mathsf{query\_onboard}() \wedge \mathit{boat}.\mathsf{query\_pos}() \hat{=} \mathit{place}\}} \mathit{value}(\tau, g.\mathsf{query\_size}())$$

Then, if *totalmis* is the total number of missionaries in a certain location (or in boats at that location), then either this must be zero or it must be greater than the total number of cannibals.

dep **DisableInherited**(CANGROUP, eat_missionaries)
acc $[t]$ $\neg$override(*cangroup*.class, eat_missionaries, CANGROUP) $\wedge$
cangroup.query_position() $\hat{=}$ *place* $\wedge$
totalmis = **people_at**$(t, \text{MISGROUP}, \mathit{place})$ +
people_in_boats_near$(t, \text{MISGROUP}, \mathit{place}) \rightarrow$
totalmis = $0 \vee$
totalmis >= **people_at**$(t, \text{CANGROUP}, \mathit{place})$ +
people_in_boats_near$(t, \text{CANGROUP}, \mathit{place})$ □

7.7 Missionaries

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

class MISGROUP *extends* GROUP

Accepted for publication in Artificial Intelligence Journal, 2003

7.8 Boat

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

class BOAT *extends* OBJECT
  attr BOAT.onboard : PLACE

There are two new methods:

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

dep **DisableInherited**(BOAT, boat_limit)
dep $[t]$ ¬override($boat$.class, boat_limit, BOAT) $\rightarrow$
  people_at(t, GROUP, *value*(t, $boat$.query_onboard())) ≤ 2 □

Mutator move_to(BANK): The move_to method is a low-level mutator that moves the boat to another BANK. This involves altering the pos attribute, but also removing the connection from the boat to its current location as well as adding a new connection from the boat to its new location.

dep **DisableInherited**(BOAT, move_to)
dep $[t]$ ¬override($boat$.class, move_to, BOAT) $\wedge$
    $boat$.move_to($bank$) $\wedge$
    $boat$.query_pos() = $oldbank$ $\rightarrow$
  Call($t+1$, $boat$.query_onboard().remove_connection($oldbank$)) $\wedge$
  Call($t+1$, $boat$.set_pos($bank$)) $\wedge$
  Call($t+1$, $boat$.query_onboard().add_connection($bank$)) □

7.9 Setting Up the Problem Instance

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:

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

This completes the modeling of the basic Missionaries and Cannibals Domain. In the next section we will describe 19 elaborations of this domain, and in Section 9, we will show how to solve the problems within the logic.

8 Elaborations of the Missionaries and Cannibals Domain

McCarthy [2] considers 19 different elaborations of the basic Missionaries and Cannibals domain, and discusses the requirements these domains place on a formalism used for modeling them and on a system for reasoning about and solving the problems. These elaborations will now be modeled in TAL-C using the object-oriented model of the MCP domain as a basis. The relations between the elaborations are shown in Figure 2.

The elaborations are often 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. We concentrate on the modeling of the domains rather than on the computational properties of a reasoner finding plans for problem instances or proving that no plan exists. However, we do feel that most of the main points of the domain elaborations have been modeled in a reasonable manner.

Earlier versions of the domain definitions are available as part of the VITAL tool, which can be downloaded from the web [17]. The current versions will be added in the next release of VITAL.

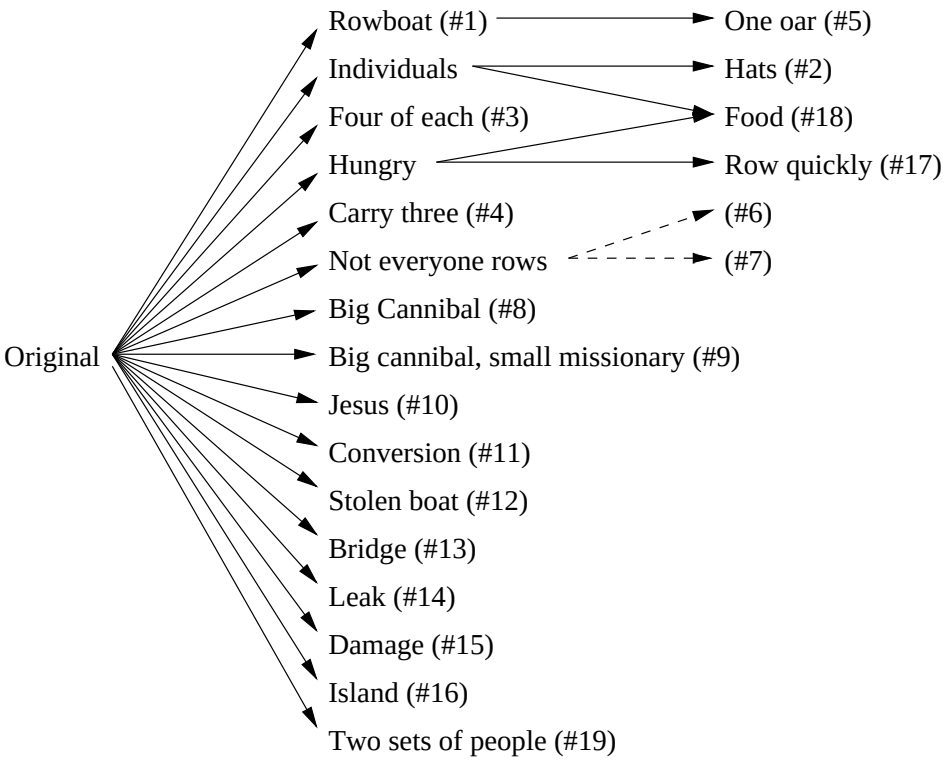

Fig. 2. Elaborations of the Missionaries and Cannibals domain

8.1 Domain and Problem Specifications

We will consider each problem to consist of two parts. The *domain specification* defines the classes being used together with their attributes and the inheritance hierarchy, while the *problem specification* 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 $\mathcal{L}(\mathrm{ND})$ domain specification is removed or modified in any of the elaborations.

Although it would have been possible to use similar techniques to model the problem specification in Section 7.9 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.

8.2 The Boat Is a Rowboat (#1)

In the first elaboration by McCarthy, we find out that the boat is in fact a rowboat. This requires a new class ROWBOAT, subclass of BOAT, and **vera** must be made an instance of ROWBOAT.

However, no new information is given regarding rowboats. The elaborated scenario is essentially similar to the original problem – with the important exception that if further information about rowboats is presented in the future, we will be able to draw additional or different conclusions about **vera**.

class ROWBOAT *extends* BOAT
  obj **vera** : ROWBOAT

8.3 Missionaries and Cannibals Have Hats (#2)

In the second elaboration, the missionaries and cannibals have hats, all different. The hats may be exchanged among the missionaries and cannibals.

8.3.1 Viewing Missionaries and Cannibals as Individuals

While missionaries and cannibals used to be interchangeable and could be modeled as groups, they must now be seen as individuals. A class for persons is added, together with a group attribute that keeps track of the group to which the person belongs. This attribute should be initialized to suitable values in the initial state.

class PERSON *extends* OBJECT
  attr PERSON.group : GROUP

What remains is ensuring that a person always belongs to the right group. The only method moving people between groups is GROUP.modify_group(), but this method only specifies how many people should move to another group, not *which* people should move. Adapting this method to a model containing individuals may seem to be a quite complicated task, and it might even seem like this elaboration is beyond the capabilities of our logic. Fortunately, this is not the case.

The solution lies in making the group attribute dynamic – allowing it to vary freely over time without a persistence assumption – and then constraining it using a new addition to modify_group(). The additional constraint essentially states that if n people should move from $group_1$ to $group_2$, then there should be exactly n individuals who previously belonged to $group_1$ and now instead belong to $group_2$. Note that we do not *override* modify_group: We merely add to its previous definition.

Constraint modify_group(GROUP$_2$, n): Suppose that at some timepoint, the method $group_1$.modify_group($group_2, n$) is invoked for two different groups $group_1$ and $group_2$.

The definition of this method in the superclass (Section 7.5) states that if n is positive, then n people should move from $group_1$ to $group_2$. This means that exactly n individuals that used to belong to $group_1$ should now belong to $group_2$. This is achieved using the first method implementation below.

On the other hand, if n is negative, then $-n$ people should move in the other direction. But in this case there must also be a method call $group_2$.modify_group ($group_1, -n$), according to the original constraints on modify_group in Section 7.5. Since $-n$ is positive, this case is also handled by the first method implementation below.

$$\begin{array}{l}\text{acc } [t]\ \neg\text{override}(group_1.\text{class}, \text{modify\_group}, \text{GROUP}) \wedge \\ \quad [t+1]\ group_1.\text{modify\_group}(group_2, n) \wedge \\ \quad n \geq 0 \wedge \\ \quad group_1 \neq group_2 \rightarrow \\ \quad \sum_{\{p \mid p \in \text{PERSON} \wedge [t]\ p.\text{group} \hat{=} group_1 \wedge [t+1]\ p.\text{group} \hat{=} group_2\}} 1 = \max(0, n)\end{array}$$

On the other hand, if for some timepoint t and some some distinct pair of groups $group_1$ and $group_2$, the method is not invoked at all (for any n), then no person at all should move from $group_1$ to $group_2$. The rule above does not guarantee this, since if the method is not invoked at all for a certain pair of groups, the antecedent of the implication cannot hold. An additional method implementation is required, which is used when the method is *not* called:

$$\begin{array}{l}\text{acc } [t]\ \neg\text{override}(group_1.\text{class}, \text{modify\_group}, \text{GROUP}) \wedge \\ \quad [t+1]\ \neg\exists n[group_1.\text{modify\_group}(group_2, n)] \wedge \\ \quad group_1 \neq group_2 \rightarrow \\ \quad \sum_{\{p \mid p \in \text{PERSON} \wedge [t]\ p.\text{group} \hat{=} group_1 \wedge [t+1]\ p.\text{group} \hat{=} group_2\}} 1 = 0\end{array}$$

Note that the final line could also be written as follows:

$$\neg\exists person[[t]\ person.\text{group} \hat{=} group_1 \wedge [t+1]\ person.\text{group} \hat{=} group_2]$$

These two method implementations are sufficient to extend the group model into a model with individuals, together with a new problem instance definition where six PERSON objects are declared and placed into the groups on the left bank. This hybrid group/individual model is admittedly somewhat more complex than a pure individual-based model, but it is nevertheless interesting to see that the model can be adjusted in this way without having to remove or completely rewrite existing classes and methods.

It should be noted that this implementation makes it impossible to move $n \geq 0$ people from *group* to $group_2$ and at the same time move $n' \geq 0$ people from *group*

to *group*$_2$, where $n \neq n'$. Although one could possibly interpret this to mean that $n + n'$ people move from *group* to *group*$_2$, this would only introduce complications that are generally unnecessary.

8.3.2 Hats

Given the domain presented above, where the missionaries and cannibals are seen as individuals, adding hats and the possibility to exchange them is trivial. A new class for hats is added, together with a new hat attribute for determining which hat belongs to which person:

class HAT *extends* OBJECT
  attr PERSON.hat : HAT

Accessor and mutator methods for the hat attribute are added. Also, a method for exchanging hats is added to PERSON:

Mutator exchange_hats(PERSON): Exchange hats with the given person.

dep **DisableInherited**(PERSON, exchange_hats)
dep $[t]$ ¬override(*person*.class, exchange_hats, PERSON) ∧
  person.exchange_hats(*person'*) →
  Call$(t+1,$ *person*.set_hat(*value*$(t,$ *person'*.get_hat()))) ∧
  Call$(t+1,$ *person'*.set_hat(*value*$(t,$ *person*.get_hat())))

Finally, six hats must be created and the hat attribute must be initialized.

8.4 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. The problem specification is therefore modified accordingly:

obs $[0]$ **cleft**.pos $\hat{=}$ **left** ∧ **cleft**.size $\hat{=}$ 4
obs $[0]$ **mleft**.pos $\hat{=}$ **left** ∧ **mleft**.size $\hat{=}$ 4
obs . . .

Accepted for publication in Artificial Intelligence Journal, 2003

8.5 The Boat Can Carry Three (#4)

In the fourth elaboration, the boat can carry three people, while 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, it will be done the right way:

class SIZEBOAT *extends* BOAT
  attr SIZEBOAT.capacity : `Integer`

Constraint boat_limit(): Ensure that the capacity is not exceeded.

dep **DisableInherited**(SIZEBOAT, boat_limit)
acc $[t]$ ¬override(*sizeboat*.class, boat_limit, SIZEBOAT) →
  people_at(t, GROUP, *value*(t, *sizeboat*.query_onboard())) ≤
  value(t, *sizeboat*.query_capacity()) □

Using the capacity attribute it is now possible to model boats with arbitrary limits on the number of passengers.

8.6 One Oar on Each Bank (#5)

Suppose that the boat is a rowboat, and that there is initially one oar on each bank. Suppose also that one person can cross the river with a single oar, but that two people will need both oars to cross together.

Modeling this as an extension of elaboration 1 requires a new class for oars, and two oars must be created and placed in their initial positions. These oars can later be moved between connected positions using set_pos().

class OAR *extends* OBJECT
  obj **oar1**, **oar2** : OAR
  obs $[0]$ **oar1**.pos $\hat{=}$ **left**
  obs $[0]$ **oar2**.pos $\hat{=}$ **right**

It is also necessary to ensure that the boat only moves when a sufficient number of oars are available. One person can row using one oar, and two persons can row using two oars – in other words, the number of people in the boat must not exceed the number of oars.

dep **DisableInherited**(ROWBOAT, oar_limit)
acc $[t]$ ¬override(*rowboat*.class, oar_limit, ROWBOAT) ∧
rowboat.query_onboard() $\hat{=}$ *place* →
people_at$(t,$ GROUP$,$ *place*$) \leq \sum_{o \mid o \in \text{OAR} \wedge [t]\ o.\text{query\_pos}() \hat{=} place} 1$

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

In elaboration 6, only one cannibal and one missionary can row (which leaves the problem solvable), while in elaboration 7, no missionary can row (which makes it unsolvable). These elaborations extend elaboration 1 (the rowboat). Two new classes for rowing cannibals and rowing missionaries are introduced, and the problem initialization is changed accordingly (for example, six new groups are added):

class ROWCANGROUP *extends* CANGROUP
class ROWMISGROUP *extends* MISGROUP
obj **rcleft**, **rcvera**, **rcright** : ROWCANGROUP
obj **rmleft**, **rmvera**, **rmright** : ROWMISGROUP
obs . . .

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

dep **DisableInherited**(BOAT, row_limit)
acc $[t]$ ¬override(*boat*.class, row_limit, BOAT) ∧
boat.query_pos() $\hat{\neq}$ *value*$(t+1,$ *boat*.query_pos()) →
people_at$(t,$ ROWCANGROUP$,$ *boat*.query_onboard()) + **people_at**
$(t,$ ROWMISGROUP$,$ *boat*.query_onboard()) > 0

8.8 Big Cannibal (#8)

In the eighth elaboration, one cannibal is too big to fit into the boat with another person. A new group class for big cannibals is introduced, and the problem specification is changed accordingly:

class BIGCANGROUP *extends* CANGROUP
obj **bcleft**, **bcvera**, **bcright** : BIGCANGROUP
obs . . .

A new constraint method is added to this class, to ensure that if any big cannibals are on board a boat, then there is exactly one person on board that boat:

dep **DisableInherited**(BIGCANGROUP, size_limit)
acc $[t]$ ¬override(*bigcangroup*.class, size_limit, BOAT) ∧
people_at$(t,$ BIGCANGROUP, *boat*.query_onboard$()) > 0 \rightarrow$
people_at$(t,$ GROUP, *boat*.query_onboard$()) \mathrel{\hat{=}} 1$

8.9 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.

To model this elaboration, 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.

class SMALLMISGROUP *extends* MISGROUP
class BIGCANGROUP *extends* CANGROUP
dep **DisableInherited**(BIGCANGROUP, eat_small)
acc $[t]$ ¬override(*bigcangroup*.class, eat_small, BIGCANGROUP) ∧
people_at$(t,$ BIGCANGROUP$, place) = 1 \wedge$
people_at$(t,$ SMALLMISGROUP$, place) = 1 \rightarrow$
people_at$(t,$ GROUP$, place) > 2$

8.10 Jesus (#10)

One of the missionaries is Jesus Christ, who can walk on water. A new group class is created, and objects are instantiated and initialized for each position:

class JESUSGROUP *extends* MISGROUP
obj **jleft**, **jvera**, **jright** : JESUSGROUP
obs . . .

The query_can_move_to() method from Section 7.5 is then overridden with a variation that does not require the origin and the destination to be connected.

Accessor query_can_move_to(JESUSGROUP′): Jesus objects can move between non-connected places (that is, cross the river without a boat).

dep **DisableInherited**(JESUSGROUP, query_can_move_to)
dep $[t]$ ¬override(*jesusgroup*.class, move_persons, JESUSGROUP) ∧
jesusgroup.class $\mathrel{\hat{=}}$ *jesusgroup′*.class) $\rightarrow$
Set(*jesusgroup*.query_can_move_to(*jesusgroup′*) $\mathrel{\hat{=}}$ true) □

Accepted for publication in Artificial Intelligence Journal, 2003

8.11 Conversion (#11)

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

dep **DisableInherited**(MISGROUP, convert)
dep $[t]$ ¬override(*misgroup*.class, convert, MISGROUP) ∧
people_at$(t,$ MISGROUP$,$ *place*$) \geq 3 \wedge$
people_at$(t,$ CANGROUP$,$ *place*$) = 1 \rightarrow$
Call$(t+1,$ *misgroup*.modify_group$($*misgroup*$, 1)) \wedge$
Call$(t+1,$ *misgroup*.modify_group$($*cangroup*$, -1))$

This elaboration takes advantage of the true concurrency in TAL-C [5]. For example, modify_group automatically handles situations where a cannibal is boarding a boat while another is being converted to a missionary.

8.12 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-C 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 BOAT.prob_not_stolen : `Real`
obs $\forall$*boat*.$[0]$*boat*.prob_not_stolen $\hat{=}$ 1.0

dep **DisableInherited**(BOAT, update_prob)
dep $[t]$ ¬override(*boat*.class, update_prob, BOAT) ∧ *boat*.query_onboard
$() \hat{=}$ *place* ∧
people_at$(t,$ GROUP$,$ *place*$) = 1 \wedge$
people_at$(t,$ CANGROUP$,$ *place*$) = 1 \rightarrow$
Set$([t+1]$ *boat*.prob_not_stolen $\hat{=} 0.9 *$ *value*$(t,$ *boat*.prob_not_stolen$))$

8.13 The Bridge (#13)

There is a bridge. The capacity of the bridge is not specified, but as long as at least two people can cross simultaneously, an arbitrary number of people can cross. Add a BRIDGE class and ensure that its capacity limit is respected.

class BRIDGE *extends* PLACE
attr BRIDGE.capacity : `Integer`
dep **DisableInherited**(BRIDGE, bridge_limit)
acc $[t]$ ¬override($bridge$.class, bridge_limit, BRIDGE) $\rightarrow$ **people_at** $(t, \text{GROUP}, bridge) \leq value(t, bridge.\text{query\_capacity}())$

Then instantiate a bridge, provide it with a capacity and connect it to the left and right banks.

8.14 The Boat Leaks (#14)

In elaboration 14, the boat leaks and must be bailed. Add a new durational boolean attribute bailed with default value false. The intention is that bailing the boat at a specific timepoint makes bailed true at that timepoint. A constraint method requires that the boat always be bailed (but does not cause the boat to be bailed – the user, or the controller, must call the bail method).

attr BOAT.bailed : `boolean`
dep **DisableInherited**(BOAT, bail)
dep $[t]$ ¬override($boat$.class, bail, BOAT) $\rightarrow I([t]\ boat.\text{set\_bailed}(\text{true}))$
dep **DisableInherited**(BOAT, must_bail)
acc $[t]$ ¬override($boat$.class, must_bail, BOAT) $\rightarrow [t]\ boat.\text{query\_bailed}()$

8.15 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 at the destination. We also add a temporal constant *crosstime* representing the amount of time required to cross the river.

class SLOWBOAT *extends* BOAT
attr SLOWBOAT.emergency : BOOLEAN
obj **onriver** : BANK

The move_to method, which is responsible for moving the boat to another BANK, must also be overridden and split into two parts: (1) move the boat to **onriver**, and (2) after *crosstime* timepoints, if there has been no emergency, move it to the desired bank. The second part takes advantage of TAL-C's ability to handle delays [20,12].

Mutator move_to(BANK): Move the boat to another bank, with a delay.

Accepted for publication in Artificial Intelligence Journal, 2003

dep **DisableInherited**(SLOWBOAT, move_to)
dep $[t]$ ¬override(*slowboat*.class, move_to, SLOWBOAT) ∧
slowboat.move_to(*bank*) ∧
slowboat.query_pos() = *oldbank* →
Call$(t+1,$ *slowboat*.query_onboard().remove_connection(*oldbank*)) ∧
Call$(t+1,$ *slowboat*.set_pos(**onriver**))

dep $[t]$ ¬override(*slowboat*.class, move_to, SLOWBOAT) ∧
slowboat.move_to(*bank*) ∧
$[t+1, t+$ *crosstime*$]$ ¬*slowboat*.query_emergency() →
Call$(t+$ *crosstime*, *slowboat*.set_pos(*bank*)) ∧
Call$(t+$ *crosstime*, *slowboat*.query_onboard().add_connection(*bank*)) □

If there is an emergency, the second dependency constraint above will not be triggered, and the boat will not end up at its intended destination. Instead, the boat should move to the left bank and be repaired.

Constraint emergency_behavior: If there are people on board and repairs are necessary, automatically move to the left bank for repairs.

dep **DisableInherited**(SLOWBOAT, emergency_behavior)
dep $[t]$ ¬override(*slowboat*.class, emergency_behavior, SLOWBOAT) ∧
slowboat.query_emergency() ∧
people_at$(t,$ BOAT, *slowboat*.query_onboard()) > 0 →
Call$(t+1,$ *slowboat*.set_pos(**left**)) ∧
Call$(t+1,$ *place*.add_connection(**left**)) ∧
Call$(t+1,$ *slowboat*.set_emergency($\bot$))]

□

8.16 The Island (#16)

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

obj **island** : BANK

8.17 Four Cannibals, Four Missionaries, Row Quickly (#17)

Elaboration 17 is defined as follows by McCarthy:

There are four cannibals and four missionaries, but if the strongest of the missionaries rows fast enough, the cannibals won't have gotten so hungry that they will eat the missionaries. This could be made precise in various ways, but the information is usable even in vague form.

First, two new group classes are introduced: One for strong missionaries, and one for cannibals that may or may not be hungry. The necessary instances are created and initialized.

class HCANGROUP *extends* CANGROUP
class STMISGROUP *extends* MISGROUP
obj **hcleft**, **hcvera**, **hcright** : HCANGROUP
obj **smleft**, **smvera**, **smright** : STMISGROUP
obs ...

A new boolean attribute is introduced to keep track of whether the cannibals in a certain group are hungry or not. In the initial state, nobody is hungry.

attr HCANGROUP.hungry : `boolean`
obs $\forall$*hcangroup*.[0] *hcangroup*.hungry $\hat{=}$ false

The old eat_missionaries constraint stated unconditionally that the missionaries must never be outnumbered by the cannibals in any location. This constraint must be weakened slightly: If none of the cannibals at a certain location are hungry, it does not matter whether the missionaries are outnumbered or not.

dep **DisableInherited**(HCANGROUP, eat_missionaries)
acc $[t]$ $\neg$override(*hcangroup*.class, eat_missionaries, HCANGROUP) $\wedge$
hcangroup.query_position() $\hat{=}$ *place* $\wedge$
hcangroup.query_hungry() $\wedge$
totalmis = **people_at**(t, MISGROUP, *place*) +
people_in_boats_near(t, MISGROUP, *place*) $\rightarrow$
totalmis = 0 $\vee$
totalmis >= **people_at**(t, HCANGROUP, *place*) +
people_in_boats_near(t, HCANGROUP, *place*) □

What remains is determining exactly when the cannibals should become hungry. The information given by McCarthy could be interpreted in many different ways. It would be possible to model the strength of each person, let the amount of time required to cross the river depend on the strength of the rowers, and let every cannibal become hungry at, say, time 10. Although this could be modeled in TAL-C, we choose a simpler interpretation where the cannibals immediately become hungry when the strong missionary is no longer in the boat.

Accepted for publication in Artificial Intelligence Journal, 2003

dep **DisableInherited**(HCANGROUP, become_hungry)
dep $[t]$ ¬override(*hcangroup*.class, become_hungry, HCANGROUP) ∧
$t \geq 1 \wedge$
people_at$(t$, STMISGROUP, *boat*.query_onboard$()) < 1 \rightarrow$
Call$(t+1$, *hcangroup*.set_hungry(true))

8.18 Four Cannibals, Four Missionaries, Food (#18)

Like in the previous elaboration, there are four missionaries and four cannibals, and the cannibals are initially not hungry. The difference is that in this elaboration, the missionaries have some food that they can give to the cannibals whenever they become hungrier. As McCarthy notes, this requires comparing a situation and a successor situation, which is clearly not a problem in TAL-C.

This is a quite complex elaboration. Since the level of hunger cannot be associated with a group, it requires treating people as individuals, and we will use elaboration 2 as the starting point. To this we will have to add a way of determining when to feed the cannibals, and keep track of how hungry they are and how much food each missionary has.

We begin by creating the subclasses FOODCANGROUP and FOODMISGROUP, in which some new methods will be added and others will be overridden. We also need the classes MISSIONARY and CANNIBAL, subclasses of PERSON (which was inherited from elaboration 2).

class FOODCANGROUP *extends* CANGROUP
class FOODMISGROUP *extends* MISGROUP
class MISSIONARY *extends* PERSON
class CANNIBAL *extends* PERSON

obj **cleft**, **cvera**, **cright** : FOODCANGROUP
obj **mleft**, **mvera**, **mright** : FOODMISGROUP
obj **misA**, **misB**, **misC**, **misD** : MISSIONARY
obj **canA**, **canB**, **canC**, **canD** : CANNIBAL

Cannibals can have different levels of hunger, modeled as an integer attribute. Missionaries have a certain amount of food. This must be initialized at time zero, and arbitrary numbers have been used below.

attr CANNIBAL.hunger : `Integer`
attr MISSIONARY.food : `Integer`
obs $[0]$ **canA**.hunger $\hat{=} 1 \wedge$ **canB**.hunger $\hat{=} 0 \wedge$
canC.hunger $\hat{=} 0 \wedge$ **canD**.hunger $\hat{=} 0$
obs $[0]$ **misA**.food $\hat{=} 3 \wedge$ **misB**.food $\hat{=} 1 \wedge$
misC.food $\hat{=} 7 \wedge$ **misD**.food $\hat{=} 7$

Accepted for publication in Artificial Intelligence Journal, 2003

The feed method feeds a cannibal a certain amount of food. As in the modify_group method, two dependency constraints sum the arguments of all concurrent method invocations.

dep **DisableInherited**(MISSIONARY, feed)

dep $[t]$ ¬override(*missionary*.class, feed, MISSIONARY) $\rightarrow$
Set($[t+1]$*missionary*.food $\hat{=}$ *value*$(t,$ *missionary*.food$)-$
$$\sum_{\{\langle c,x\rangle \mid c\in \text{CANNIBAL}\wedge[t]\ \textit{missionary}.\text{feed}(c,x)\}} x$$

dep $[t]$ ¬override(*missionary*.class, feed, MISSIONARY) $\rightarrow$
Set($[t+1]$*cannibal*.hunger $\hat{=}$ *value*$(t,$ *cannibal*.hunger$)+$
$$\sum_{\{\langle m,x\rangle \mid m\in \text{MISSIONARY}\wedge[t]\ m.\text{feed}(\textit{cannibal},x)\}} x$$

If a cannibal is becoming hungrier, the missionaries may or may not feed him.

dep **DisableInherited**(MISSIONARY, do_feed)
dep $[t]$ ¬override(*missionary*.class, do_feed, MISSIONARY) $\wedge$
missionary.query_group() $\hat{=}$ *foodmisgroup* $\wedge$
cannibal.query_group() $\hat{=}$ *foodcangroup* $\wedge$
foodmisgroup.query_pos() $\hat{=}$ *foodcangroup*.query_pos() $\wedge$
$[t+1]$ *cannibal*.query_hunger() $>$ *value*$(t,$ *cannibal*.query_hunger()) $\rightarrow$
$\exists n.0 \leq n \leq 1 \wedge$ **Call**$(t+2,$ *missionary*.feed(*cannibal*$, n))$

The cannibals must become hungrier now and then. For example, they might become hungrier at time 2 and 4:

dep $t=2 \vee t=4 \rightarrow$ **Set**$([t+1]$ *cannibal*.hunger $\hat{=}$ *value*$(t,$ *cannibal*.hunger $)+1$

Finally, the original eat_missionaries constraint stated unconditionally that the missionaries must never be outnumbered by the cannibals in any location. Again, this constraint must be weakened slightly: If none of the cannibals at a certain location has a hunger level greater than 2, it does not matter whether the missionaries are outnumbered or not.

dep **DisableInherited**(FOODCANGROUP, eat_missionaries)
acc $[t]$ ¬override(*foodcangroup*.class, eat_missionaries, FOODCANGROUP) $\wedge$
foodcangroup.query_position() $\hat{=}$ *place* $\wedge$
($\exists$*cannibal*.*cannibal*.get_group() $\hat{=}$ *foodcangroup* $\wedge$
cannibal.get_hunger() > 2) $\wedge$
totalmis = **people_at**$(t,$ FOODMISGROUP$,$ *place*$)+$
people_in_boats_near$(t,$ FOODMISGROUP$,$ *place*$) \rightarrow$
totalmis $= 0 \vee$
totalmis $>=$ **people_at**$(t,$ FOODCANGROUP$,$ *place*$)+$
people_in_boats_near$(t,$ FOODCANGROUP$,$ *place*$)$ □

Accepted for publication in Artificial Intelligence Journal, 2003

8.19 Two Sets of People (#19)

In the final elaboration, there are two sets of missionaries and cannibals too far apart along the river to interact. A new attribute same_set keeps track of which banks belong to the same "set", and must be initialized using observation statements:

attr BANK.same_set(BANK) : `boolean`
obs $[0]$ **left**.same_set(**right**) $\wedge \ldots$

The following constraint method ensures that the origin and destination are in the same set.

dep **DisableInherited**(BOAT, move_same_set)
dep $[t]$ $\neg$override($boat$.class, move_same_set, BOAT) $\rightarrow$
$boat$.query_pos().query_same_set($value(t+1, boat$.query_pos()))

8.20 Classes in the Elaborated Missionaries and Cannibals Problems

In the elaborations presented above we created a number of new classes that extend the class hierarchy shown in Figure 1. An overview of the new class hierarchy is shown in Figure 3.

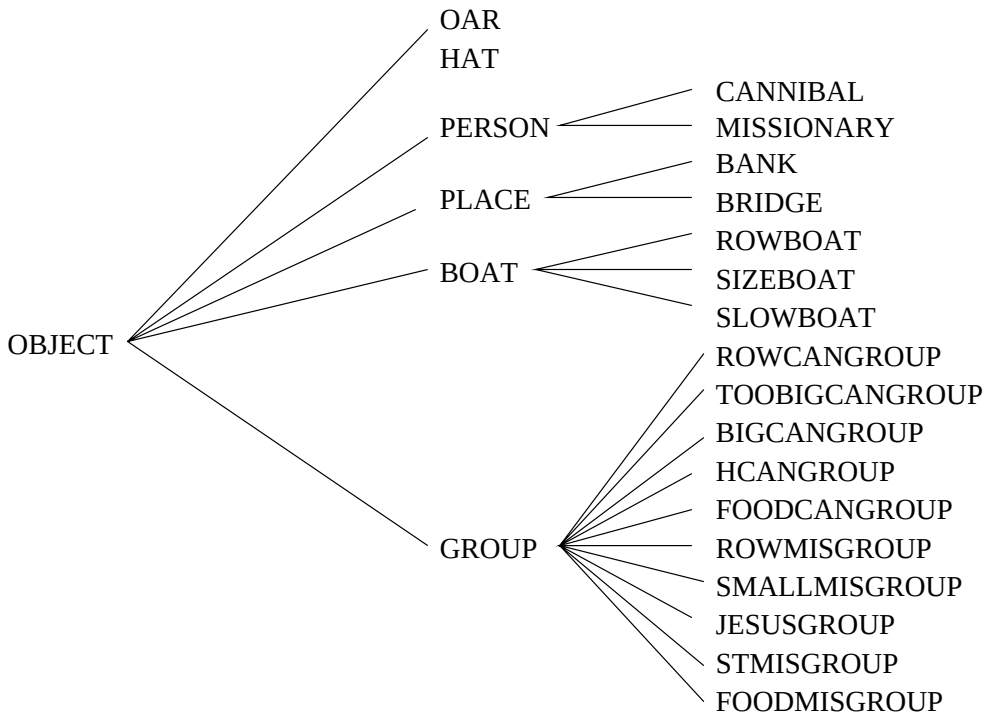

Fig. 3. Classes in the Elaborated Missionaries and Cannibals Problems

Accepted for publication in Artificial Intelligence Journal, 2003

9 Solving the Missionaries and Cannibals Problems

Though the main focus of this article is on modeling, we would also like to actually *solve* the Missionaries and Cannibals problem instances presented by McCarthy. In other words, given that the missionaries and cannibals are located on the left river bank, a suitable set of actions (or method invocations) should be found that moves everyone to the right bank without any missionaries being eaten.

Although one could use the model only for prediction and then apply standard planning algorithms to solve each problem, we instead choose to build on the ideas for automatic control presented in [21] and model a controller within the logic. Since the different elaborations have slightly different demands on the controller, it will be modeled as another class whose methods can be overridden in subclasses, providing another test of the elaboration tolerance of the object-oriented approach.

The main idea behind the controller is that whenever there is a choice between different actions that could be invoked, this choice is modeled using an incompletely specified constraint method. For example, whenever a boat can move, a constraint method in the controller will call the boat's set_pos method to move it, but the exact destination will not be specified.

Every logical model of the resulting narrative corresponds to a different set of actions that could potentially be taken by the missionaries and cannibals, given that the cannibals never outnumber the missionaries in any location as required by eat_missionaries() (Section 7.6). What remains is choosing a model that actually achieves the goal, rather than just containing missionaries and cannibals moving around randomly. To achieve this, we assume (like Lifschitz [19]) that we know the length t_* of the plan to be generated. By constraining the state at time t_* to be a solution state, where everyone is at the right river bank, we ensure that any remaining logical model must correspond to a valid plan. [5] The value t_* is made available in the narrative as a temporal constant, and will be used in some of the controller methods.

For the original problem, we know that the minimal plan length is 12. The plan lengths for the 19 elaborations will be shown together with the timing results in Section 9.3, and the goals must of course also be altered for those elaborations that involve different group types or a larger number of missionaries and cannibals.

obs $t_* = 12$
obs $[t_*]$ **mright**.size $\hat{=} 3 \wedge$ **cright**.size $\hat{=} 3$

[5] 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.

9.1 A Controller for the Original Problem

The controller for the original problem will consist of a class CONTROLLER with a set of constraint methods defined below. One instance must be created in every elaboration.

class CONTROLLER *extends* OBJECT
  obj **ctrl** : CONTROLLER

9.1.1 Allowing People to Move

The first step in defining the controller is allowing people to move randomly between groups in connected locations. This is done by adding the following method:

Constraint move_persons(): Moves an unspecified number of people (possibly zero) between compatible groups in connected locations, where the compatibility is tested using the query_can_move_to method. For example, if there is a group of cannibals $group_1$ on the left bank and a group of cannibals $group_2$ on the boat, and the boat is at the left bank (the places are connected), then cannibals may move between $group_1$ and $group_2$. Note that GROUPs never move – people move by changing the size of two groups. Also note that the number of people moving from $group_1$ to $group_2$ can naturally be equal to zero.

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

dep **DisableInherited**(CONTROLLER, move_persons)
dep $[t]$ $\neg$override($controller$.class, move_persons, CONTROLLER) $\wedge$
  $group_1$.query_can_move_to($group_2$) $\rightarrow$
  $\exists n$ $[-value(t, group_2.\text{query\_size}()) \leq n \wedge n \leq value(t, group_1.\text{query\_size}()) \wedge$
    Call$(t+1, group_1.\text{modify\_group}(group_2, -n)) \wedge$
    Call$(t+1, group_2.\text{modify\_group}(group_1, n))]$ □

9.1.2 Allowing Boats to Move

The second step consists of forcing the boat to move to another randomly selected bank whenever anyone is onboard. The following method is added to BOAT:

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.

dep **DisableInherited**(CONTROLLER, move_boat)
dep $[t]$ ¬override(*controller*.class, move_boat, CONTROLLER) ∧
people_at$(t,$ GROUP, *value*$(t,$ *boat*.query_onboard$()))) > 0 \rightarrow$
$\exists$*bank*$[[t]$ *boat*.query_pos$() \not\doteq$ *bank* ∧
Call$(t,$ *boat*.move_to(*bank*$))]$ □

9.1.3 Additional Control: Don't be Stupid

In addition to the nondeterministic choice of actions provided by the methods above, it is also possible to introduce some more "intelligence" in the controller by adding further constraints on the acceptable state sequence.

There is no point in allowing a state to repeat.

Constraint no_repetitions(): At each timepoint, at least one group should change sizes.

dep **DisableInherited**(CONTROLLER, no_repetitions)
acc $[t]$ ¬override(*controller*.class, no_repetitions, CONTROLLER) $\rightarrow$
$\exists$*group*.*value*$(t,$ *group*.query_size$()) \neq$ *value*$(t+1,$ *group*.query_size$())$ □

There should be at least one person on the boat, except at the first and last timepoint in the plan. This avoids plans where everyone leaves the boat but nobody else boards it, leaving it empty for a period of time.

Constraint boat_not_empty(): There should be someone on the boat.

dep **DisableInherited**(CONTROLLER, boat_not_empty)
acc $[t]$ ¬override(*controller*.class, boat_not_empty, CONTROLLER) $\rightarrow$
$$\forall t.t > 0 \wedge t < t_* - 1 \rightarrow \sum_{\substack{\{g \mid g \in \text{GROUP} \wedge \\ [t]\ g.\text{query\_pos}() \doteq \textbf{onvera}\}}} \textit{value}(t, g.\text{query\_size}()) > 0$$

□

9.2 Additions for the Elaborations

Although the controller presented above is sufficient for the original version of the Missionaries and Cannibals domain, some of the elaborations alter basic properties of the domain and require further elaborations of the controller.

9.2.1 One Oar on Each Bank (#5)

In the fifth elaboration, there is one oar on each bank. To solve this problem, a cannibal must row alone to the other bank, pick up the second oar, and then row back. This means that there must be an interval of time where no groups change sizes, so no_repetitions must be modified in a new controller class OARCONTROLLER: If there is an oar in a position near the rowboat, then no groups have to change. An instance of OARCONTROLLER should then be created instead of an instance of CONTROLLER.

class OARCONTROLLER *extends* CONTROLLER
  obj **ctrl** : OARCONTROLLER

Constraint no_repetitions(): At each timepoint, at least one group should change sizes.

dep **DisableInherited**(OARCONTROLLER, no_repetitions)
acc $[t]$ ¬override(*oarcontroller*.class, no_repetitions, OARCONTROLLER) $\rightarrow$
  $\exists oar.[t+1]oar$.query_pos().query_connection(*rowboat*.query_onboard()) $\vee$
  $\exists group.value(t, group$.query_size()) $\neq$ *value*$(t+1, group$.query_size()) $\square$

In addition to this relaxation of no_repetitions, it is also necessary to extend the controller to take an oar whenever one is available.

Constraint take_oars(): If a rowboat is at a river bank where an oar is available, then the oar should be moved into the boat.

dep **DisableInherited**(OARCONTROLLER, take_oars)
dep $[t]$ ¬override(*oarcontroller*.class, take_oars, OARCONTROLLER) $\wedge$
    oar.query_pos() $\hat{=}$ *rowboat*.query_pos() $\rightarrow$
  Call$(t+1, oar$.set_pos(*rowboat*.query_onboard())) $\square$

9.2.2 The Bridge (#13)

If there is a bridge, the boat does not necessarily have to be used at all timepoints. The boat_not_empty constraint has to be disabled, which is done by overriding it in a new controller subclass BRIDGECONTROLLER without providing a new implementation.

class BRIDGECONTROLLER *extends* CONTROLLER
  obj **ctrl** : BRIDGECONTROLLER
  dep **DisableInherited**(BRIDGECONTROLLER, boat_not_empty)

9.2.3 The Boat Leaks (#14)

If the boat can leak, the controller must be extended to call the bail action at all timepoints.

class BAILCONTROLLER *extends* CONTROLLER
  obj **ctrl** : BAILCONTROLLER

  dep **DisableInherited**(BAILCONTROLLER, do_bail)
  dep $[t]$ ¬override(*bailcontroller*.class, do_bail, BAILCONTROLLER) →
    Call(t, *bailboat*.bail())

9.2.4 The Boat Can Be Damaged (#15)

In elaboration 15, the boat can be damaged, and the action of moving to another river bank had to be split into two events: Moving to the river, and then after *crosstime* timepoints, arriving at the destination. The original controller states that groups must always change sizes from t to $t + 1$, which clearly cannot be the case in this scenario. Instead, the groups must change sizes from time t to time $t + crosstime$, unless there was an emergency.

class SLOWCONTROLLER *extends* CONTROLLER
  obj **ctrl** : SLOWCONTROLLER

  dep **DisableInherited**(SLOWCONTROLLER, no_repetitions)
  acc $[t]$ ¬override(*slowcontroller*.class, no_repetitions, SLOWCONTROLLER) ∧
    $[t + 1, t + crosstime - 1]$¬*slowboat*.query_emergency() →
    ∃*group*.*value*(t, *group*.query_size()) ≠ *value*($t + 1$, *group*.query_size())

An additional precondition is required for move_boat: The controller should not call move_to for a boat when that boat is on the river.

  dep **DisableInherited**(SLOWBOAT, move_boat)
  dep $[t]$ ¬override(*slowcontroller*.class, move_boat, SLOWCONTROLLER) ∧
      boat.query_pos() ≠ **onriver** ∧
    people_at(t, GROUP, *value*(t, *boat*.query_onboard())) > 0 →
    ∃*bank*[$[t]$ *boat*.query_pos() ≠ *bank* ∧
        Call(t, *boat*.move_to(*bank*))]

9.3 Results

The timings in Table 1 were generated by the research tool VITAL [17] using Java 1.3.1 and the HotSpot Server virtual machine on an 1800 MHz Pentium 4 machine. The total number of time steps in each plan is shown (including one step for initialization) together with the total amount of time required for generating the plan.

Times are specified in seconds. We also provide some comparisons with the 10 elaborations implemented by Lifschitz [19] in the Causal Calculator [22], 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. This is especially true because (at least in VITAL) timings depend very much on the exact formulation of an elaboration, and could change drastically simply by altering the order in which objects are declared.

Two of the problems were unsolvable. We have not proved this within the logic: The logic-based controller used to solve the remaining 17 problems is not a full planner, and like the Causal Calculator, it requires as input the length of the plan to be generated. Proving that no plan (of arbitrary length) would solve these two problem instances would require additional reasoning outside the logic.

10 Traffic World

The object-oriented framework presented in this article has also been used for modeling the Traffic World scenario proposed in the Logic Modeling Workshop [6], previously modeled by Henschel and Thielscher [23] using the Fluent Calculus [24]. This domain consists of cars moving in a road network represented as a graph structure, together with a TAL-C controller class that "drives" a car. A complete TAL-C action scenario will soon be available at the VITAL web page [17].

11 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 [25,26], 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 [27].

Elaboration	Steps	Time (VITAL)	Time (CC)
Original	12	1.5	17.6
1	12	1.5	–
2	12	6.5	–
3	Unsolvable		
4	12	2.8	18
5	14	2.5	44
6	14	5.2	273
7	Unsolvable		
8	16	11.3	9746
9	12	7.8	22
10	6	1.7	–
11	12	2.3	55
12	12	1.8	–
13	5	1.6	2
14	12	1.7	9
15	36	5.2	–
16	16	165.5	1894
17	10	3.8	7361
18	14	24.0	–
19	12	16.6	–

Table 1
Test Results for the Missionaries and Cannibals Problems

The modeling methodology presented in this article uses a different kind of 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++ [28], 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-C, 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 [29,30], 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 [31] 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.

12 Conclusions

This article has presented a way to do object-oriented modeling in an existing logic of action and change, allowing large domains to be modeled in a more systematic way and providing 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 few new macros have been introduced in this article, 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 $\mathcal{L}(\mathrm{FL})$ more efficient, although the current version of VITAL does not take advantage of this.

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] J. de Kleer, J. S. Brown, A qualitative physics based on confluences, Artificial Intelligence 24 (1–3) (1984) 7–83.

[2] J. McCarthy, Elaboration tolerance, in: The 1998 Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense-98), London, 1998, available at http://www.dcs.qmul.ac.uk/research/krr/events/CS98/CS14.ps.

[3] M. Abadi, L. Cardelli, A Theory of Objects, Springer Verlag, 1996, see http://www.luca.demon.co.uk/TheoryOfObjects.html.

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

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

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

[7] P. Doherty, J. Gustafsson, L. Karlsson, J. Kvarnström, TAL: Temporal Action Logics – language specification and tutorial, Electronic Transactions on Artificial Intelligence 2 (3–4) (1998) 273–306, available at http://www.ep.liu.se/ej/etai/1998/009/.

[8] E. Sandewall, Features and Fluents: A Systematic Approach to the Representation of Knowledge about Dynamical Systems, Vol. 1, Oxford University Press, 1994.

[9] P. Doherty, Reasoning about action and change using occlusion, in: A. G. Cohn (Ed.), Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94), John Wiley and Sons, 1994, pp. 401–405, available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/ecai94.ps.gz.

[10] J. Gustafsson, P. Doherty, Embracing occlusion in specifying the indirect effects of actions, in: L. C. Aiello, J. Doyle, S. C. Shapiro (Eds.), Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR-96), Morgan Kaufmann Publishers, San Francisco, 1996, pp. 87–98, available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/final-kr96.ps.gz.

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

[12] L. Karlsson, J. Gustafsson, P. Doherty, Delayed effects of actions, in: H. Prade (Ed.), Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98), John Wiley and Sons, 1998, pp. 542–546.

[13] J. Kvarnström, P. Doherty, TALplanner: A temporal logic based forward chaining planner, Annals of Mathematics and Artificial Intelligence 30 (2000) 119–169.

[14] P. Doherty, J. Kvarnström, TALplanner: A temporal logic-based planner, AI Magazine 22 (3) (2001) 95–102.

[15] J. McCarthy, Circumscription – a form of non-monotonic reasoning, Artificial Intelligence 13 (1980) 27–39, reprinted in [32].

[16] P. Doherty, W. Łukaszewicz, Circumscribing features and fluents: A fluent logic for reasoning about action and change, in: D. M. Gabbay, H. J. Ohlbach (Eds.), Proceedings of the 1st International Conference on Temporal Logic (ICTL-94), Vol. 827 of Lecture Notes in Computer Science, Springer Verlag, 1994.

[17] J. Kvarnström, VITAL. An on-line system for reasoning about action and change using TAL, available at http://www.ida.liu.se/~jonkv/vital/ (1997–2003).

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

[19] V. Lifschitz, Missionaries and cannibals in the causal calculator, in: Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR-2000), Morgan Kaufmann Publishers, 2000, pp. 85–96.

[20] P. Doherty, J. Gustafsson, Delayed effects of actions = direct effects + causal rules, Linköping Electronic Articles in Computer and Information Science 3, available at http://www.ep.liu.se/ea/cis/1998/001.

[21] J. Gustafsson, Extending temporal action logic, Ph.D. thesis, Linköping Studies in Science and Technology, Dissertation No. 689 (2001).

[22] N. McCain, the Texas Action Group, The causal calculator, available at http://www.cs.utexas.edu/users/tag/cc/.

[23] A. Henschel, M. Thielscher, The LMW traffic world in the fluent calculus, available at http://www.ida.liu.se/ext/etai/lmw/ (1999).

[24] M. Thielscher, Introduction to the fluent calculus, Electronic Transactions on Artificial Intelligence 2 (3–4) (1998) 179–192, available at http://www.ep.liu.se/ej/etai/1998/006/.

[25] A. Borgida, R. Brachman, D. McGuinness, L. Resnick, CLASSIC: A structural data model for objects, in: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland Oregon, 1989, pp. 58–67.

[26] R. Brachman, R. Fikes, H. Levesque, KRYPTON: A functional approach to knowledge representation, Computer 16 (1983) 67–73.

[27] A. Artale, E. Franconi, A temporal description logic for reasoning about actions and plans, Journal of Artificial Intelligence Research Vol 9 (1998) 463–506.

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

[29] E. Amir, Object-oriented first-order logic, Electronic Transactions on Artificial Intelligence 3 (1999) 63–84, available at http://www.ep.liu.se/ej/etai/1999/008/.

[30] E. Amir, (De)Composition of situation calculus theories, in: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), AAAI Press / The MIT Press, 2000, pp. 456–463, available at http://www.cs.berkeley.edu/~eyal/papers/oo-sitcalc-aaai00.ps.

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

[32] J. McCarthy, Formalization of common sense, papers by John McCarthy edited by V. Lifschitz, Ablex, 1990.

A Macros in $\mathcal{L}(\mathbf{ND})$

The following is a subset of the macros used in the TAL-C version of $\mathcal{L}(\mathrm{ND})$.

A *fixed fluent formula* $[\tau]\, f \mathrel{\hat{=}} v$ expresses the fact that the fluent f has the value v at the timepoint τ. For boolean fluents, the shorthand notation $[\tau]\, f \stackrel{\text{def}}{=} [\tau]\, f \mathrel{\hat{=}} \mathsf{true}$ and $[\tau]\, \neg f \stackrel{\text{def}}{=} [\tau]\, f \mathrel{\hat{=}} \mathsf{false}$ is allowed. Boolean connectives are allowed within a temporal scope (for example, $[\tau]\, f \mathrel{\hat{=}} v \wedge g \mathrel{\hat{=}} v'$), and closed, open, or semi-open intervals are permitted (for example, $[\tau, \tau']\, f \mathrel{\hat{=}} v$).

The function $\mathit{value}(\tau, f)$ returns the value of the fluent f at the timepoint τ, where $[\tau]\, f \mathrel{\hat{=}} v$ iff $\mathit{value}(\tau, f) = v$. The expression $[\tau]\, f \mathrel{\hat{=}} g$, where f and g are fluents, is a shorthand notation for $[\tau]\, f \mathrel{\hat{=}} \mathit{value}(\tau, g)$.

An *occlusion expression* $X([\tau, \tau']\, \phi)$ expresses the fact that all fluents in ϕ are occluded (exempt from persistence or default value assumptions) in $[\tau, \tau']$. A *reassignment expression*, $R([\tau, \tau']\, \phi) \stackrel{\text{def}}{=} X([\tau, \tau']\, \phi) \wedge [\tau']\, \phi$, also requires ϕ to hold at the end of the interval, while a *durational reassignment expression* $I([\tau, \tau']\, \phi) \stackrel{\text{def}}{=} X([\tau, \tau']\, \phi) \wedge [\tau, \tau']\, \phi$ requires ϕ to hold throughout the interval. This is generalized to open, semi-open and singleton intervals. In this article, the I macro is also denoted by the more intuitive name **Set**.

An *atomic expression* is either any of the expressions defined above or a *feature*, *value* or *timepoint equality expression* ($f = f'$, $v = v'$, $\tau = \tau'$), a *temporal relational expression* ($\tau \otimes \tau'$, where $\otimes$ is a relation symbol in the temporal base structure), or an *action occurrence expression* ($[\tau, \tau']\, A(\overline{\omega})$, stating that the action A, with arguments $\overline{\omega}$, is invoked during the interval $[\tau, \tau']$).

Statements in $\mathcal{L}(\mathrm{ND})$ are formed from atomic 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.

B The Base Language $\mathcal{L}$(FL)

In order to reason about a particular narrative, it is first mechanically translated into the base language $\mathcal{L}$(FL), an order-sorted classical first-order language with equality. The base language uses the following predicates:

- *Holds*(τ, f, v) – the fluent f has the value v at time τ,
- *Occlude*(τ, f) – the fluent f may change value at τ (corresponding to the reassignment macros),
- *Occurs*(τ, τ', a) – the action a occurs between τ and τ',
- *Per*(τ, f) – the fluent f is persistent at time τ, and
- *Dur*(τ, f, v) – f has default value v at time τ.

The translation function *Trans* : $\mathcal{L}$(ND) $\rightarrow$ $\mathcal{L}$(FL) is defined in Doherty et al. [7]. Although a complete understanding of the translation is not strictly necessary, we nevertheless provide a few example translations below. The complete translation of the hiding turkey narrative from Section 2.2 will also be shown below.

- *Trans*$([\tau]\ f \hat{=} v) \stackrel{\text{def}}{=}$ *Holds*(τ, f, v)
- *Trans*$([\tau]\ f \hat{=} v \wedge g \hat{=} w) \stackrel{\text{def}}{=}$ *Holds*$(\tau, f, v) \wedge$ *Holds*(τ, g, w)
- *Trans*$([\tau, \tau']\ f \hat{=} v) \stackrel{\text{def}}{=} \forall t.\tau \leq t \leq \tau' \rightarrow$ *Holds*(t, f, v)
- *Trans*$(X([\tau]\ f \hat{=} v)) \stackrel{\text{def}}{=}$ *Occlude*(τ, f)
- *Trans*$(X((\tau, \tau']\ f \hat{=} v)) \stackrel{\text{def}}{=} \forall t.\tau < t \leq \tau' \rightarrow$ *Occlude*(t, f)
- *Trans*$([\tau, \tau']\ A(\overline{\omega})) =$ *Occurs*$(\tau, \tau', A(\overline{\omega}))$

The logical theory which is the result of the translation is still under-constrained in the sense that a number of implicit assumptions about fluent change in the world remain to be characterized. In general, we want to encode the blanket assumption that fluent values do not change unless there is a good reason for this to happen. There are a number of legitimate reasons for fluents to change value, such as action occurrences where the effects of the action change fluent values, or causal dependencies between fluents where changes in some fluents force changes in others. In TAL-C, all such legitimate reasons for change are represented implicitly using the reassignment macros R, I and X in dependency constraints and action type definitions. When translated, these statements result in constraints on the *Occlude* predicate.

In the logical theory, we want to formally encode the assumption that these are the *only* reasons for fluents to be occluded. This is done by using filtered circumscription [16], a special form of circumscription [15] where the *Occlude* predicate is circumscribed relative to the action definitions and dependency constraints with all other predicates fixed, and *Occurs* is circumscribed relative to the action occurrence formulas with all other predicates fixed. The results are combined and filtered with the $\mathcal{L}$(FL) translations of the persistence statements (forcing persistent and durational fluents to adhere to the persistence or default value assumptions), domain

constraints, observations, and timing constraints, as well as the $\mathcal{L}(\mathrm{FL})$ foundational axioms and temporal structure axioms (TAL-C uses a linear, discrete time structure with non-negative time). The resulting second-order theory can be translated into a logically equivalent first-order theory, which is then used to reason about the narrative. In the remainder of the article, $\mathit{Trans}^{+}(\mathcal{N})$ will denote the result of translating the narrative $\mathcal{N}$ into $\mathcal{L}(\mathrm{FL})$ and applying this filtered circumscription policy, which is formally defined in [7].

B.1 The Hiding Turkey Scenario in $\mathcal{L}(FL)$

The translation of the Hiding Turkey Scenario into $\mathcal{L}(\mathrm{FL})$ is somewhat more complex than its $\mathcal{L}(\mathrm{ND})$ formalization, demonstrating some of the advantages of providing the macros in $\mathcal{L}(\mathrm{ND})$. (Here, we have simplified $\neg\mathit{Holds}(\tau, f, \mathsf{true})$ into $\mathit{Holds}(\tau, f, \mathsf{false})$.)

per1 $\forall t\,[\mathtt{true} \rightarrow \mathit{Per}(t+1, \mathsf{alive}) \wedge \mathit{Per}(t+1, \mathsf{deaf}) \wedge \mathit{Per}(t+1, \mathsf{hiding}) \wedge \mathit{Per}(t+1, \mathsf{loaded})]$

per2 $\forall t\,[\mathtt{true} \rightarrow \mathit{Dur}(t, \mathsf{noise}, \mathsf{false})]$

obs1 $\mathit{Holds}(0, \mathsf{alive}, \mathsf{true}) \wedge \mathit{Holds}(0, \mathsf{loaded}, \mathsf{false}) \wedge \mathit{Holds}(0, \mathsf{hiding}, \mathsf{false})$

dep1 $\forall t\,[\mathit{Holds}(t, \mathsf{hiding}, \mathsf{false}) \wedge \mathit{Holds}(t, \mathsf{deaf}, \mathsf{false}) \wedge \mathit{Holds}(t, \mathsf{noise}, \mathsf{true})) \rightarrow \mathit{Holds}(t+1, \mathsf{hiding}, \mathsf{true}) \wedge \mathit{Occlude}(t+1, \mathsf{hiding})]$

dep2 $\forall t\,[\forall t'[t \leq t' \leq t+9 \rightarrow \mathit{Holds}(t', \mathsf{hiding}, \mathsf{true}) \wedge \mathit{Holds}(t', \mathsf{noise}, \mathsf{false})] \rightarrow \mathit{Holds}(t+10, \mathsf{hiding}, \mathsf{false}) \wedge \mathit{Occlude}(t+10, \mathsf{hiding})$

acs1 $\mathit{Occurs}(t_1, t_2, \mathsf{Load}) \rightarrow (\mathit{Holds}(t_2, \mathsf{loaded}, \mathsf{true}) \wedge \mathit{Occlude}(t_2, \mathsf{loaded}) \wedge \forall t[t_1 < t \leq t_2 \rightarrow \mathit{Holds}(t, \mathsf{noise}, \mathsf{true})] \wedge \forall t[t_1 < t \leq t_2 \rightarrow \mathit{Occlude}(t, \mathsf{noise})])$

acs2 $\mathit{Occurs}(t_1, t_2, \mathsf{Fire}) \rightarrow ((\mathit{Holds}(t_1, \mathsf{loaded}, \mathsf{true}) \wedge \mathit{Holds}(t_1, \mathsf{hiding}, \mathsf{false}) \rightarrow \mathit{Holds}(t_2, \mathsf{alive}, \mathsf{false}) \wedge \mathit{Occlude}(t_2, \mathsf{alive})) \wedge (\mathit{Holds}(t_1, \mathsf{loaded}, \mathsf{true}) \rightarrow \mathit{Holds}(t_2, \mathsf{loaded}, \mathsf{false}) \wedge \mathit{Occlude}(t_2, \mathsf{loaded})))$

occ1 $\mathit{Occurs}(1, 4, \mathsf{Load})$

occ2 $\mathit{Occurs}(5, 6, \mathsf{Fire})$

B.2 Circumscription of Occlude in the Hiding Turkey Scenario

The circumscription of the *Occlude* predicate in the action schemas (acs) and dependency constraints (dep) in the Hiding Turkey Scenario is equivalent to the following set of first-order formulas:

$$\forall t[\mathit{Occlude}(t, \mathsf{alive}) \leftrightarrow t = 6 \wedge \mathit{Holds}(5, \mathsf{loaded}, \mathsf{true}) \wedge \mathit{Holds}(5, \mathsf{hiding}, \mathsf{false})]$$

$$\forall t[\mathit{Occlude}(t, \mathsf{loaded}) \leftrightarrow t = 4 \vee t = 6 \wedge \mathit{Holds}(5, \mathsf{loaded}, \mathsf{true})]$$

Accepted for publication in Artificial Intelligence Journal, 2003

$\forall t[\neg \mathit{Occlude}(t, \mathsf{deaf})]$

$\forall t[\mathit{Occlude}(t, \mathsf{hiding}) \leftrightarrow$
$\quad \exists t'[t = t' + 1 \wedge \mathit{Holds}(t', \mathsf{hiding}, \mathsf{false}) \wedge \mathit{Holds}(t', \mathsf{deaf}, \mathsf{false}) \wedge \mathit{Holds}(t', \mathsf{noise}, \mathsf{true})] \vee$
$\quad \exists t'[t = t' + 10 \wedge \forall \tau[t' \leq \tau \leq t' + 9 \rightarrow \mathit{Holds}(\tau, \mathsf{hiding}, \mathsf{true}) \wedge \mathit{Holds}(\tau, \mathsf{noise}, \mathsf{false})]]]$

$\forall t[\mathit{Occlude}(t, \mathsf{noise}) \leftrightarrow 2 \leq t \leq 4]$

The circumscription of the *Occurs* predicate in the action occurrence statements (occ) in the Hiding Turkey Scenario is equivalent to the following first-order formula:

$\forall t, t', a[\mathit{Occurs}(t, t', a) \leftrightarrow (t = 1 \wedge t' = 4 \wedge a = \mathsf{Load}) \vee (t = 5 \wedge t' = 6 \wedge a = \mathsf{Fire})]$