Object-Oriented Reasoning about Action and Change

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

Abstract

As the scope of logics of action and change continues to increase and powerful research tools are developed, it becomes possible to model larger and more complex scenarios. Unfortunately the scenarios become harder to read and difficult to modify and debug with increasing size and complexity. These problems have been overlooked in the action and change community due to the fact that only smaller toy problems are considered. Sound modeling methodology is as essential as the primitives of the modeling language. The object-oriented paradigm is one structuring mechanism that alleviates these problems and provides a systematic means of scenario construction.

The topic of this paper is to demonstrate how many ideas from the object orientation paradigm can be used when reasoning about action and change, we show this by integrating the technique directly in an existing logic of action and change without any modification to the underlying logical language or semantics.

1 Introduction

The action and change community has primarily used toy examples as benchmarks for testing the semantic adequacy of formalisms. Most of the time, action scenarios in the literature can be described in words using a couple of sentences and the logic representation is seldom more than a page long, with the sentences grouped together by type rather than structure. These toy examples are used in order to highlight or explain some particular point the author wants to make. However, with some of the classical problems totally or partially solved, and with powerful tools available for reasoning about action scenarios, it is now both possible and necessary to model larger, more complex domains.

When we cease modeling toy domains and begin working with more complex examples, one thing becomes apparent: There is a lack of a methodology for handling large scenarios. There are no principles of good form, like the "No Structure in Function" principle from the qualitative reasoning community [7].

The following are some questions that have to be answered in order to develop a systematic means of handling intricate domains:

- Design: How do we design large domain descriptions in a consistent way?
- Elaboration tolerance: How do we, in a convenient way, modify facts in a domain description to take account of new phenomena or changed circumstances?
- Modularity: How can we group facts together in such a way that it is easy to locally change things without having to modify the entire domain description?
- Incrementality: How do we extend an already existing domain description?
- Reusability: How do we reuse parts of old domain description? How do we provide support for building libraries that provide a way of reusing often occurring structures?

These questions make it evident that there is a need for a framework or a methodology that provides the tools for modeling larger domains. The object-oriented paradigm (for example [1, 4]) does this by providing a more direct mapping to the way we think about reality in an intuitive manner.

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

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

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

The topic of this paper is to show that most object-oriented ideas can be directly modeled in our temporal action logic (TAL-C) and that they can be used as a scenario structuring mechanism for supporting the construction of larger scenarios.

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 rules that have to be satisfied whenever the method is invoked. This means that we can invoke methods over intervals of time and that several methods might be invoked concurrently.

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

2 TAL-C: Temporal Action Logic

In this section, we briefly introduce TAL-C [8, 9], which will be used as a basis for a proposal for modeling some aspects of object-orientation. The basic approach we use for reasoning about action and change (RAC) is as follows. First, represent a narrative in the surface language $\mathcal{L}(\mathrm{ND})$ which is a high-level language for representing observations, action descriptions and action occurrences, dependency constraints, domain constraints, and timing constraints about actions and their duration. Second, translate $\mathcal{L}(\mathrm{ND})$ into the base language $\mathcal{L}(\mathrm{FL})$ which is an order-sorted first-order language with four predicates $\mathit{Occlude}(t, f)$, $\mathit{Holds}(t, f, v)$, $\mathit{Per}(f)$, and $\mathit{Dur}(f, v)$, where t, f, and v are variables for time-point, fluent, and value expressions, respectively. *Holds* expresses what value a fluent has at each time-point. *Occlude* expresses that a fluent is exempt from the default assumption at a time-point. Each fluent has to be characterized as either a durational fluent, $\mathit{Dur}(f, v)$, with default value v, or a persistent fluent $\mathit{Per}(f)$, but not both. The idea is that unless a durational fluent is occluded at a time-point, it will retain its default value, while a persistent fluent at $t + 1$ retains whatever value it has at t unless it is occluded.

A linear discrete time structure is used in TAL-C. The minimization policy is based on the use of filtered preferential entailment [15] where action descriptions (**acs-**) and dependency constraints (**dep-**) are circumscribed with *Occlude* minimized and *Holds* fixed. The result is then filtered with two nochange axioms, the observations, and some foundational axioms such as unique names and temporal structure axioms.

The translation from $\mathcal{L}(\mathrm{ND})$ to $\mathcal{L}(\mathrm{FL})$ is straightforward and the reader is referred to [8, 9] for details concerning translation and the logic used.

Before we describe the operators of $\mathcal{L}(\mathrm{ND})$, we consider how to define fluent value domains.

- The simplest way of defining a domain is by directly listing its elements. A simple example of this is **dom** boolean $= \{\top, \bot\}$
- A more dynamical way of defining domains is by first describing the relation between the domains and then adding the elements. An example of

this way of defining fluent value domains would look like the following:

> **dom** THING
> **dom** VEHICLE *isa* THING
> **obj agent1** *instanceof* THING
> **obj helicopter1** *instanceof* VEHICLE

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

The $\mathcal{L}$(ND) language contains several operators for expressing different types of change. This paper, however, only describes fixed fluent formulas, occlusion assignment and exceptional assignment. The other operators are not needed in the examples discussed here and are therefore omitted.

A *fixed fluent formula* has the form $[t]f = v$ and is true if and only if the fluent f has the value v at time-point t. For boolean fluents we can use the shorthand notation $[t]f$ or $[t]\neg f$. Fixed fluent formulas do not cause change, they are used to check the value of a fluent at a given time-point.

The *Release* operator stands for *occlusion assignment* and has the form $\mathit{Release}([t]f)$[1]. The intended meaning of this statement is that the default behavior of f need not hold at time-point t. Technically this is done simply by occluding f at time-point t which means that the value of f varies freely. Occlusion assignment can be used to model non-determinism. An example of occlusion assignment is $[t]\mathit{shake} \rightarrow \mathit{Release}([t]\mathit{break})$ where *shake* and *break* are persistent boolean fluents. If *shake* is true at time-point t, then *break* is released from its persistence assumption at t and might become either true or false.

The last operator, *Set*, stands for *exceptional assignment* [2] and has the form $\mathit{Set}([t]f \mathrel{\hat{=}} v)$. Like the occlusion assignment, f is released from the default behavior but in addition we require f to take value v. For example, assume that *flow* is a durational fluent with default value 0.0. The exceptional assignment $\mathit{Set}([t]\mathit{flow}(\mathbf{tank1}, \mathbf{pipe2}) \mathrel{\hat{=}} 2.0)$ states that the fluent $\mathit{flow}(\mathbf{tank1}, \mathbf{pipe2})$ takes the value 2.0 at time-point t. Unless something else influences $\mathit{flow}(\mathbf{tank1}, \mathbf{pipe2})$, it will revert its default value 0.0 at time-point $t + 1$. If instead *flow* had been a persistent fluent, it would have kept the value 2.0 at time-point $t + 1$.

The description of TAL-C is very brief, since the purpose of this paper is not to extend or modify the logic but to show how the object-oriented paradigm can be used to succinctly structure large axiomatic theories in TAL-C.

[1] *Release* was denoted by X in previous work.
[2] *Set* was denoted by I in previous work.

3 Modeling Object-orientation in TAL-C

As has been shown elsewhere [9, 10, 11], TAL-C is a flexible and fine-grained language suitable for handling a wide class of domains. The intention of this paper is to show how many aspects of object-orientation can be used in the TAL-C language as a structuring mechanism for domain descriptions, thereby supporting the modeling of more complex domains and the reuse of parts of old domain descriptions when modeling related domains.

The object-orientation can be represented directly in the TAL-C surface language $\mathcal{L}$(ND). The versatility of durational fluents makes it straightforward to model many of the non-monotonic aspects of the object-oriented paradigm. Although some of the constructions may seem cumbersome, it is possible to introduce a new set of macros in $\mathcal{L}$(ND) to hide them.

Due to page limitations, we demonstrate the technique using a small application to watertanks; but we emphasize that the approach targets larger, more complex application domains, some of which have been successfully formalized using these techniques. We begin with two classes called TANK and FLOWTANK. The only interaction possible with the TANK class is to set the volume in the tank to a specified number. The FLOWTANK class has the same behavior as the TANK class but in addition it is possible to say that there is a flow into or out of the tank.

3.1 Classes

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

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

dom TANK *isa* OBJECT
  obj **tank1** *instanceof* TANK,

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

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

dom FLOWTANK *isa* TANK
  obj **tank2** *instanceof* FLOWTANK

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

3.1.1 Fields in an object

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

For example, our TANK class needs a *volume* field. Assuming we have a floating point value domain `float`, the *volume* field can be modeled as a fluent $\mathit{volume}(\text{TANK}) : \texttt{float}$ taking a water tank as an argument. Clearly, since FLOWTANK is a subsort of TANK, any FLOWTANK will also have a volume. In other words, the field is automatically inherited by subclasses of TANK.

The FLOWTANK class also needs a *flow* field, which can be modeled as a fluent $\mathit{flow}(\text{FLOWTANK}) : \texttt{float}$ taking a flow tank as an argument. Tanks that are not flowtanks will not be valid arguments to this new fluent, but any object of a subclass of FLOWTANK will have a *flow* field.

3.1.2 Methods

We define three types of methods: procedures, functions and constraint methods. Procedures are used to change the internal state of an object and have no return values, functions do not cause any change but have a return value, and constraint methods represent rules or constraints that must hold at all time-points.

Procedures. The only legal interaction between objects is by method invocations. In our approach, method invocations are modeled using fluents. For each class c with the corresponding fluent value domain DOMAIN, and for each procedure m we wish to define in that class with arguments of sorts $\langle s_1, \ldots, s_n \rangle$, we define a durational fluent $m(\text{OBJECT}, \text{DOMAIN}, s_1, \ldots, s_n) : \texttt{boolean}$ with default value false. At any time-point where an object o wants to invoke this procedure in another object o', with the actual arguments $x_1, \ldots, x_n$, it should make $m(o, o', x_1, \ldots, x_n)$ true.

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

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

$$\textbf{dep}\ \forall t, \mathit{caller} \in \textsc{object}, \mathit{self} \in \textsc{tank}, f \in \texttt{float}$$
$$[t]\mathsf{set\text{-}volume}(\mathit{caller}, \mathit{self}, f) \rightarrow \mathit{Set}([t]\mathit{volume}(\mathit{self}) \hat{=} f)$$

This dependency constraint states that if any object *caller* calls the set-volume procedure in the tank *self* with argument f, then the volume in *self* becomes f.

Since all objects created from subclasses of TANK by necessity are members of the TANK domain, this method can be invoked on all of them. The above example models a public procedure, which means that any other object can invoke it. If we want to make the method protected, which means that it is only invokable by subclasses, the domain of *caller* can simply be set to TANK instead of OBJECT.

Functions. The second type of method is functions. Functions are used to get values from objects. Instead of representing the invocation with a boolean durational fluent, as the procedures, we let the invocation fluent be a dynamic fluent that has the same domain as the return value. The value of the fluent is bound by the body of the function, as in:

$$\textbf{dep}\ \forall t, \mathit{caller} \in \textsc{object}, \mathit{self} \in \textsc{tank}$$
$$[t]\mathsf{query\text{-}volume}(\mathit{caller}, \mathit{self}) = [t]\texttt{volume}(\mathit{self}))$$

Note that functions are not allowed to contain any *Set* or *Release* macros.

Constraint methods. In contrast to normal object-oriented programming, some types of behavior have to be active at all time-points. For this we introduce a special type of method that we call constraint methods. A constraint method looks just like a method, but we do not require any invocation fluent to be true in order for the method to be active. An example of this is the flow in a flowtank. The changing of the volume does not depend on any method invocation; it should automatically be done at every time-point. The constraint method for this would look like the following:

$$\textbf{dep}\ \forall t, \mathit{self} \in \textsc{flowtank}, f_1, f_2 \in \texttt{float}$$
$$([t]\mathit{flow}(\mathit{self}) = f_1 \wedge [t]\mathit{volume}(\mathit{self}) = f_2) \rightarrow$$
$$\mathit{Set}([t+1]\mathsf{set\text{-}volume}(\mathit{self}, \mathit{self}, f_1 + f_2)),$$

where *flow* is the inflow of the tank minus the outflow of the tank. This constraint means that if at time-point t we have flow f_1 and volume f_2 then we invoke the set-volume method with argument $f_1 + f_2$ at time-point $t + 1$.

3.2 Overriding

Another useful feature of object-orientation is the ability to override methods. A method defined in a superclass may be defined again in a subclass, and this new definition takes precedence over the old definition. Say, for example that we later want to use the FLOWTANK class but we want to have an upper limit (10 units of water) to the amount that we can put into the tank. In this case we construct a new class called OFTANK (OverFlow TANK) to represent such tanks. This class extends the FLOWTANK class with a new method description for the set-volume method that overrides the old behavior. For this to work in our approach, two things have to be done when a method is defined. First, we have to ensure that all methods with the same name higher in the class-tree are not invokable. Secondly, we have to make sure that for each object, the method is invokable only if no subclass overrides it. To do these things we introduce a new durational fluent called *override*`(object,method,classname)` which normally is false, to represent that for a given object `object`, the method `method` defined in class `classname` is overridden.

The first step is done by adding a statement of the following form each time a method is defined:

$$\begin{aligned}&\mathbf{dep}\ \forall t, c \in \texttt{classnames}, i \in \textsc{currentclass} \\ &\qquad [t]\mathit{subclass}(\textsc{currentclass}, c) \rightarrow \mathit{Set}([t]\mathit{override}(i, \mathsf{methodname}, c)),\end{aligned} \tag{1}$$

where CURRENTCLASS is the class in which the method is being defined, and methodname is the name of the method being defined, and i ranges over all instances of class CURRENTCLASS. The intended meaning of the above statement is that *override* should be true for all methods with the same name, defined in superclasses. For notational convenience we will use the macro **ClassMethod**(CURRENTCLASS, methodname) as a shorthand for statements of type (1).

The second step consists of adding, in our method definitions, the requirement that the method is not overridden for the given object in the current class. We will use the macro **Invoked**(CURRENTCLASS, methodname, $\overline{f}$) as a shorthand[3] for

$$\begin{aligned}&[t]\mathsf{methodname}(\mathit{caller}, \mathit{self}, \overline{f}) \wedge \\ &[t]\neg \mathit{override}(\mathit{self}, \mathsf{methodname}, \textsc{currentclass}).\end{aligned}$$

The set-volume method we defined earlier should instead be written in the following way, to accommodate the possibility of overriding:

$$\begin{aligned}&\mathbf{dep}_{1a}\ \mathbf{ClassMethod}(\textsc{tank}, \mathsf{set\text{-}volume}) \\ &\mathbf{dep}_{1b}\ \forall t, \mathit{caller} \in \textsc{object}, \mathit{self} \in \textsc{tank}, f \in \texttt{float} \\ &\qquad \mathbf{Invoked}(\textsc{tank}, \mathsf{set\text{-}volume}, f) \rightarrow \mathit{Set}([t]\mathit{volume}(i) \hat{=} f)]\end{aligned}$$

If a method is defined as above, it overrides all methods with the same name higher in the hierarchy. It also makes it possible to override this method if some subclass redefines the method.

[3]The **Invoked** macro is context dependent on *self* and *caller*. They could have been added as arguments but that clutters the presentation so we leave them as they are.

4 Elaboration tolerance

According to McCarthy [12], 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 of the ideas used in the object-oriented paradigm make it easy to build elaboration tolerant scenarios. This is not surprising since the reasons behind the object-oriented paradigm include modularization and the possibility to reuse code.

With inheritance it is possible to specialize a class, adding more methods and constraints.

Overriding is another powerful tool useful for increasing elaboration tolerance. It allows us to change some behaviors of a class without having to know all the details of that class. This way we do not have to do any "surgery"[4] if we want to change the behavior of a subclass. We only have to override the methods that we want to change, leaving the entire original scenario description unchanged.

5 Related work

Much work has been done in combining object-oriented ideas with the area of knowledge representation. One such area is description logics (see for example [6, 5]). Description logics are languages tailored for expressing knowledge about concepts (similar to classes) and concept hierarchies. They are usually given a Tarski style declarative semantics, which allows them to be seen as sublanguages of predicate logic. One starts with primitive concepts and roles, and can use the language constructs (such as intersection, union, role quantification, etc.) to define new concepts and roles. The main reasoning tasks are classification and subsumption checking. This means that description logic hierarchies are very dynamic and that it is possible to add new concepts or objects at runtime that are automatically sorted into the correct place in the concept hierarchy. Some work has been done in combining description logics and reasoning about action and change (see for example [3]).

The modeling methodology presented in this chapter has a very simple class hierarchy that is constructed at translation time and is thereafter static. Classes have to be explicitly positioned in the hierarchy and classes and objects cannot be constructed once the narrative has been translated. Description logics do not have class methods or explicit time, both of which are essential in the work presented here.

The approach presented in this chapter bears more resemblance to object-oriented programming languages such as Prolog++ [14], C++ or Java. In these languages, a method is a sequence of code that is procedurally executed when the method is invoked. In our approach, however, a method is a set of rules that have to be satisfied whenever the method is invoked. The fact that delays

[4]McCarthy uses the term surgery for describing the act of going through the scenario description and changing specific values by hand.

can be modeled in TAL-C means that methods can be invoked over intervals of time and that complex processes can be modeled using methods. It is also possible to invoke multiple methods concurrently.

An interesting approach to combining logic and object-orientation is Amir's object-oriented first-order logic [2], which allows a theory to be constructed as a graph of smaller theories. Each subtheory communicates with the other via interface vocabularies. The algorithms for the object-oriented first-order logic suggest that the added structure of object-orientation can be used to significantly increase the speed of theorem proving.

The work by Morgenstern [13] 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.

6 Discussion

The need for a methodology of scenario description construction becomes evident as soon as we leave toy examples behind and try to model realistic dynamic domains. As the size and complexity of domains increase, it becomes more and more difficult to read, modify and debug them.

This paper has presented a way to do object-oriented modeling in an already existing logic of action and change.

The advantage of the work presented here is that larger domains can be modeled in a more systematic way and that we can group rules together in such a way that it is possible to locally change the representation. This leads to increased reusability and elaboration tolerance.

The main difference between our work and other approaches to combining knowledge representation and object-orientation is due to the explicit timeline in TAL-C. Methods can be called over time periods or instantaneously, concurrently or with overlapping time intervals. Methods can relate to one state only or describe processes that take many time-points to complete.

Our approach also allows us to use our language for phenomena not usually modeled in sequential object-oriented languages. It is for example straightforward to sum the arguments of multiple method invocations taking place concurrently. An example of this would be to say that we allow any number of objects to set their individual flow into a tank at the same time, then compute the resulting net flow by adding together the arguments of all the invocations of the set-flow method.

The ideas presented in this paper do not require any modification of the TAL-C language or semantics, only a restriction on the surface language used to represent scenarios. In this manner, we enforce more structure on our narratives in order to get modularity and reusability. It is reasonable to believe that this can be used to make theorem proving in $\mathcal{L}(\mathrm{FL})$ more efficient. The work by Amir on an object-oriented first-order logic and its inference algorithms [2] supports the claim that object-oriented structure can be used to gain significant increases in the speed of theorem proving.

Finally the modularization also provides a nice interface for hybrid narratives in the sense that some of our objects might not be encoded in our logic. We can for example have a class for doing complex mathematics as an outside source, implemented procedurally in another object-oriented programming language. The interaction between the standard classes and the outside classes are simply handled just as method invocations, much like remote method invocations (RMI) in JAVA.

References

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

[2] E. Amir. Object-oriented first-order logic. Workshop on Nonmonotonic Reasoning, Action and Change. IJCAI 99, Aug 1999.

[3] A. Artale and E. Franconi. A temporal description logic for reasoning about actions and plans. *Journal of Artificial Intelligence Research*, Vol 9, 1998.

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

[5] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick. Classic: A structural data model for objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 1989.

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

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

[8] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnström. TAL: Temporal action logics language specification and tutorial. Linköping Electronic Articles in Computer and Information Science, 1998. Available at: http://www.ep.liu.se/ea/cis/1998/015/.

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

[10] L. Karlsson, J. Gustafsson, and P. Doherty. Delayed effects of actions. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 542–546, Aug 1998.

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

[12] J. McCarthy. Elaboration tolerance. In *Common Sense 98*, 1998.

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

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

[15] E. Sandewall. Filter preferential entailment for the logic of action and change. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence, (IJCAI-89).* Morgan Kaufmann, 1989.