# Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming*

José Meseguer

SRI International, Menlo Park, CA 94025, and
Center for the Study of Language and Information,
Stanford University, Stanford, CA 94305

**Abstract.** The *inheritance anomaly* [23] refers to the serious difficulty in combining inheritance and concurrency in a simple and satisfactory way within a concurrent object-oriented language. The problem is closely connected with the need to impose *synchronization constraints* on the acceptance of a message by an object. In most concurrent object-oriented languages this synchronization is achieved by *synchronization code* controlling the acceptance of messages by objects. Synchronization code is often hard to inherit and tends to require extensive redefinitions. The solutions that have appeared so far in the literature to alleviate this problem seem to implicitly assume that better, more reusable, mechanisms are needed to create and structure synchronization code. The approach taken in this paper is to consider the inheritance anomaly as a problem *caused* by the very presence of synchronization code. The goal is then to completely eliminate synchronization code. This is achieved by using order-sorted rewriting logic, an abstract model of concurrent computation that is machine-independent and extremely fine grain, and that can be used directly to program concurrent object-oriented systems. Our proposed solution involves a distinction between two different notions of inheritance, a type-theoretic one called *class* inheritance, and a notion called *module* inheritance that supports reuse and modification of code. These two different notions address two different ways in which the inheritance anomaly can appear; for each of them we propose declarative solutions in which no explicit synchronization code is ever used.

## 1 Introduction

The term "inheritance anomaly" has been coined by Satoshi Matsuoka and Akinori Yonezawa [23] to describe what is widely recognized as a serious difficulty in combining inheritance and concurrency in a simple and satisfactory way within a concurrent object-oriented language. Early references pointing out serious difficulties in this area include [20, 32, 35, 6].

* Supported by Office of Naval Research Contracts N00014-90-C-0086 and N00014-92-C-0518, and by the Information Technology Promotion Agency, Japan, as a part of the R & D of Basic Technology for Future Industries "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

The problem is closely connected with the need to impose *synchronization constraints* on the acceptance of a message by an object. A well-known example is a bounded buffer, where a **put** message should be accepted only if the buffer is not full, and a **get** message should be accepted only if the buffer is not empty.

In most concurrent object-oriented languages this synchronization is achieved by special code controlling how the messages will be accepted. The code performing such control is called *synchronization code*. The problem is that often this kind of synchronization code is hard to inherit and tends to require extensive redefinitions. This difficulty has been illustrated by a number of examples in the literature. Indeed, the problem is considered so thorny that a number of well-known concurrent object-oriented languages such as POOL/T [6], Act1 [22], and ABCL/1 [37] have given up supporting inheritance as a basic language feature.

A number of proposals to alleviate this problem have appeared in the literature, including [20, 35, 33, 7, 31, 36, 23, 19, 9]. It seems fair to say that, although some good progress has been made, the problem is considered far from solved. Matsuoka and Yonezawa [23] present an excellent analysis and survey of the anomaly by means of a series of increasingly more difficult examples that show where some of the proposed solutions break down. We adopt those examples in this paper to illustrate the characteristics of our own solution.

Somehow implicit in all the solutions that have appeared in the literature is the assumption that better, more reusable, mechanisms are needed to create and structure synchronization code. Indeed, what the different languages and solutions proposed so far seem to have in common is the presence of two different kinds of code, namely usual code for changing the state of an object by the reception of a message, and synchronization code to control the invocation of the usual code. The default assumption is that if no synchronization code is given the usual code can always be invoked.

The approach taken in this paper is to consider the inheritance anomaly as a problem *caused* by the very presence of synchronization code. The logical solution if we take this hypothesis seriously is to *completely eliminate* synchronization code. This is done by adopting a declarative style of programming in which the effects of messages on objects are described by logical axioms called *rewrite rules*. Each rewrite rule characterizes circumstances under which a concurrent change can take place in the system, as well as the appropriate change in such circumstances. Since change can only take place by application of rewrite rules, the appropriate conditions for the reception of messages are indeed *implicit* in the rewrite rules themselves. Therefore, no explicit synchronization code is ever needed, and the problem of how to inherit such code—which constitutes the inheritance anomaly—disappears. In this way, no difficulty remains for having a fully satisfactory integration of inheritance and concurrency in an object-oriented language. Therefore, rather than talking about *solving* the inheritance anomaly it would have been more accurate to speak of *eliminating* the anomaly.

Our proposed solution involves a distinction between two different ways in which the inheritance anomaly can appear, namely:

1. the case in which the behavior of messages previously defined in superclasses is not contradicted by their behavior in a subclass (adding a new message **get2**—to get two elements at once—in a subclass of the bounded buffer class is a typical example), and
2. the case in which the behavior of messages previously defined is in fact modified (adding a **gget** message that acts just as a **get** message, except that it cannot be accepted if the last message received was a **put** is a typical example).

These two cases can be best distinguished by introducing a precise distinction between two different notions of inheritance which, unfortunately, tend to be conflated in the common use of this term:

1. a type-theoretic one, whose purpose is the taxonomic *classification* of objects and in which the behavior of messages in a superclass is never contradicted by their behavior in a subclass (although additional behavior can be exhibited by subclasses, including the introduction of new rules, new messages, and new attributes); we call this notion *class* inheritance, and restrict the notion of *subclass* only to pairs of classes for which this relation holds;
2. a notion called *module* inheritance that supports reuse and modification of code and in which the behavior of messages previously defined in a class can indeed be modified. The key idea is to view the two *modules* in which the relevant old and new classes were introduced as standing in a (module) inheritance relation, not the classes themselves.

For each of the two cases in which the inheritance anomaly can manifest itself, these two inheritance mechanisms plus the use of rewrite rules provide a respective declarative solution in which no explicit synchronization code is ever used. The case solved by class inheritance is clearly the simplest. In the rewriting logic abstract model of concurrent computation [25, 26], the code for a class is an unstructured *set* of rewrite rules, with each rule acting independently of the others. For a subclass in our sense, this set of rules is typically enlarged by adding some new rules, but this in no way alters the previously given rules which remain exactly as before and are inherited from the superclass or superclasses.

The structure of the paper is as follows. In Section 2 the semantic framework used throughout the paper, namely rewriting logic, is introduced informally by means of examples (a precise definition of the rules of rewriting logic is given in Appendix A). In Section 3 the syntax of Maude's object-oriented modules [26] used to present the examples discussed in the paper is first used; it is shown how such modules are just sugared versions of theories in rewriting logic and their de-sugared versions are presented. In Section 4 the semantics of *class inheritance* is presented in terms of the order-sorted type structure of rewriting logic and shown to completely eliminate any anomalies that can be described in terms of such a notion inheritance. Section 5 discusses module inheritance and examples that fit within that category and shows how they can be solved in a way that does not involve any synchronization code nor, more generally, any concurrency considerations whatsoever. Section 6 recapitulates and summarizes the key characteristics of the solution that we propose. Section 7 discusses implementation issues, and Section 8 makes some concluding remarks.

## 2 Rewriting Logic as a Semantic Framework for Concurrent Object-Oriented Programming

We informally introduce rewriting logic by means of a simple example and explain how it provides a language-independent semantic framework for concurrent object-oriented programming and, more generally, for concurrent programming. A precise definition of the rules of rewriting logic is given in Appendix A; a detailed account of rewriting logic and its semantics, and of how it unifies many existing models of concurrency can be found in [25].

Rewriting logic is a logic to reason correctly about the evolution in time of a concurrent system. The distributed state of a concurrent system is represented as a *term* whose subterms represent the different components of the concurrent state. Typically, however, the *structure* of the concurrent state may have a variety of equivalent term representations because it satisfies certain *structural laws*. For example, in a concurrent object-oriented system the concurrent state, which is usually called a *configuration*, has typically the structure of a *multiset* made up of objects and messages. Therefore, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax as

```
subsorts Object Msg < Configuration .
op __ : Configuration Configuration -> Configuration
                                  [assoc comm id: null] .
```

where the multiset union operator `__` is declared to satisfy the structural laws of associativity and commutativity and to have identity `null`. The subtype declaration[2]

```
subsorts Object Msg < Configuration .
```

states that objects and messages are singleton multiset configurations, so that more complex configurations are generated out of them by multiset union.[3]

As a consequence, we can abstractly represent the configuration of a typical concurrent object-oriented system as an equivalence class $[t]$ modulo the structural laws of associativity, commutativity and identity obeyed by the multiset union operator of a term expressing a union of objects and messages, i.e., as a multiset of objects and messages.

[2] For our treatment of the inheritance anomaly it is very important to use a typed version of rewriting logic that supports subtypes; typing aspects are further explained in Sections 3 and 4.

[3] Of course, we do not want two different objects with the same name in any such configuration, but this can be easily enforced (see [26, Section 4.4]).

An *object* in a given state is also represented as a term

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where $O$ is the object's name or identifier, $C$ is its class, the $a_i$'s are the names of the object's ***attribute identifiers***, and the $v_i$'s are the corresponding ***values***. The set of all the attribute-value pairs of an object state is formed by repeated application of the binary union operator $\_,\_$ which also obeys structural laws of associativity, commutativity, and identity; i.e., the order of the attribute-value pairs of an object is immaterial.

For example, a bounded buffer whose elements are numbers can be represented as an object with three attributes: a **contents** attribute that is a list of numbers of length less than or equal to the bound, and attributes **in** and **out** that are numbers counting how many elements have been put in the buffer or got from it since the buffer's creation. For example, a typical bounded buffer state can be

```
< B : BdBuff | contents: 9 5 6 8, in: 7, out: 3 >
```

Concurrent interaction with the buffer can be achieved by means of **put** and **get** messages, with an appropriate reply message from the buffer after a **get**. We can for example assume the syntax

```
put_in_ : Nat OId -> Msg .
getfrom_replyto_ : OId OId -> Msg .
to_elt-in_is_ : OId OId Nat -> Msg .
```

for **puts**, **gets**, and replies, respectively, where **Nat** is the type of natural numbers, and **OId** is the type of object identifiers, and where in each message's syntactic form each underbar must be filled with an element of the appropriate type as indicated by the list of types after the ":" and with the entire message being of course of type **Msg**.

In rewriting logic sentences are rewrite rules of the form

$$[t] \longrightarrow [t']$$

or, more generally, conditional rewrite rules of the form

$$r : [t] \longrightarrow [t'] \;\; \textit{if} \;\; [u_1] \longrightarrow [v_1] \wedge \ldots \wedge [u_k] \longrightarrow [v_k].$$

What those sentences axiomatize are the basic *local transitions* that are possible in a concurrent system. For example, in a concurrent object-oriented system including bounded buffers that communicate through messages the local transitions of bounded buffers are axiomatized by rewrite rules of the form

```
(put E in B) < B : BdBuff | contents: Q, in: N, out: M > =>
   < B : BdBuff | contents: E Q, in: N + 1, out: M >
   if (N - M) < bound .
```

```
(getfrom B replyto I)
   < B : BdBuff | contents: Q E, in: N, out: M > =>
   < B : BdBuff | contents: Q, in: N, out: M + 1 >
   (to I elt-in B is E) .
```

where `E, N, M` range over natural numbers and `Q` over lists. The first rule specifies the conditions under which a `put` message can be accepted (namely, that `N - M` is smaller than `bound`) and the corresponding effect; the second rule does the same for `get` messages (note that the requirement that the buffer must not be empty is implicit in the pattern `Q E` for the `contents` attribute).

What the rules of deduction of rewriting logic support is sound and complete reasoning about the concurrent transitions that are possible in a concurrent system whose basic local transitions are axiomatized by given rewrite rules. That is, the sentence $[t] \longrightarrow [t']$ is provable in the logic using the rewrite rules that axiomatize the system as axioms if and only if the concurrent transition $[t] \longrightarrow [t']$ is possible in the system. A precise account of the model theory of rewriting logic fully consistent with the above system-oriented interpretation, and proving soundness, completeness, and the existence of initial models is given in [25].

The intuitive idea behind the rules of rewriting logic in Appendix A is that proofs in rewriting logic exactly correspond to concurrent computations in the concurrent system being axiomatized, and that such concurrent computation can be understood as concurrent rewritings *modulo* the structural laws obeyed by the concurrent system in question. In the case of a concurrent object-oriented system such structural laws include the associativity, commutativity and identity of the union operators `__` and `_,_`, and this means that the rules can be applied regardless of order or parentheses. For example, a configuration such as

```
(put 7 in B1) < B2 : BdBuff | contents: 2 3, in: 7, out: 5 >
< B1 : BdBuff | contents: nil, in: 2, out: 2 >
(getfrom B2 replyto C)
```

(where the buffers are assumed to have a large enough bound) can be rewritten into the configuration

```
< B2 : BdBuff | contents: 2, in: 7, out: 6 >
< B1 : BdBuff | contents: 7, in: 3, out: 2 >
(to C elt-in B2 is 3)
```

by applying concurrently the two rewrite rules[4] for `put` and `get` modulo associativity and commutativity.

Intuitively, we can think of messages as "traveling" to come into contact with the objects to which they are sent and then causing "communication events" by application of rewrite rules. In rewriting logic, this traveling is accounted for in a very abstract way by the structural laws of associativity, commutativity and identity. This abstract level supports both synchronous and asynchronous

[4] Note that rewrite rules for natural number addition have also been applied.

communication [25, 26], and provides great freedom and flexibility to consider a variety of alternative implementations at lower levels.

For the purposes of the present paper rewriting logic should be regarded as a language-independent semantic framework which, by itself, does not make any commitments to specific features or synchronization styles, and in which many different concurrent object-oriented languages could be given a precise mathematical semantics. However, to ease the exposition and make our discussions concrete, the examples used will be written in the syntax of the Maude language. Since, as we shall see, Maude modules are nothing but sugared versions of theories in rewriting logic, this does not matter much. Nevertheless, for purposes of efficient implementation Maude's language design introduces specific syntactic restrictions that are discussed in Section 7.

## 3 Maude's Object-Oriented Modules and their Translation into Rewrite Theories

We illustrate the syntax of object-oriented modules with a module for bounded FIFO buffers; we then make explicit the rewrite theory of which the module is a sugared version. We assume as previously defined a parameterized data type[5] `LIST` that is instantiated to form lists of natural numbers. We assume `bound` to be a natural number, but we do not care which one[6]

```
omod BD-BUFF is
  protecting NAT .
  protecting LIST[Nat] .
  class BdBuff | contents: List, in: Nat, out: Nat .
  initially contents: nil, in: 0, out: 0 .
  msg put_in_ : Nat OId -> Msg .
  msg getfrom_replyto_ : OId OId -> Msg .
  msg to_elt-in_is_ : OId OId Nat -> Msg .
  vars B I : OId .
  vars E N M : Nat .
  var Q : List .
  rl (put E in B) < B : BdBuff | contents: Q, in: N, out: M > =>
       < B : BdBuff | contents: E Q, in: N + 1, out: M >
       if (N - M) < bound .
  rl (getfrom B replyto I)
```

[5] Algebraic data types are also regarded as rewrite theories; their equations are assumed to be Church-Rosser and are used as rewrite rules. In Maude, such data types are declared in *functional modules* [26] and belong to a functional sublanguage very similar to OBJ3 [16]. By contrast, rules in object-oriented modules typically fail to be Church-Rosser, and in some cases may never terminate.

[6] In Maude this module should be most naturally parameterized by two parameters, namely the type of data elements and the size bound; however, to simplify the example we avoid parameterization.

```
      < B : BdBuff | contents: Q E, in: N, out: M > =>
      < B : BdBuff | contents: Q, in: N, out: M + 1 >
      (to I elt-in B is E) .
endom
```

After the keyword `class`, the name of the class—in this case `BdBuff`—is given, followed by a "`|`" and by a list of pairs of the form `a: S` separated by commas, where `a` is an attribute identifier and `S` is the type inside which the values of such an attribute identifier must range in the given class. In this example, the attributes are the `contents`, and the `in` and `out` counters. The `initially` clause states that when buffers are created they are empty and have their two counters set to 0. The messages and rewrite rules are identical to those in Section 2, but the type of the variables in the rules has now been made explicit.

We give below the essential aspects[7] of the translation of this module into a rewrite theory. Since in Maude rewrite theories themselves correspond to what are called *system modules* with keywords **mod** and **endm**, we express this translation in a system module notation.

```
mod BD-BUFF# is
  extending CONFIGURATION .
  protecting NAT .
  protecting LIST[Nat] .
  sorts <BdBuff BdBuff .
  subsort BdBuff < Object .
  subsort <BdBuff < CId .
  subsorts Nat List < Value .
  op BdBuff : -> <BdBuff .
  op put_in_ : Nat OId -> Msg .
  op getfrom_replyto_ : OId OId -> Msg .
  op to_elt-in_is_ : OId OId Nat -> Msg .
  var X : <BdBuff .
  vars B I : OId .
  vars E N M : Nat .
  var Q : List .
  var ATTS : Attributes .
  sct < B : X | contents: Q, in: N, out: M, ATTS > : BdBuff .
  rl (put E in B) < B : X | contents: Q, in: N, out: M, ATTS >
       => < B : X | contents: E Q, in: N + 1, out: M, ATTS >
       if (N - M) < bound .
  rl (getfrom B replyto I)
       < B : X | contents: Q E, in: N, out: M, ATTS > =>
       < B : X | contents: Q, in: N, out: M + 1, ATTS >
       (to I elt-in B is E) .
endm
```

[7] We omit the rewrite rules for object initialization associated with the `initially` clause; on the matter of object creation see [26, Section 4.4].

In the translation process, the most basic structure shared by all object-oriented modules is made explicit by the `CONFIGURATION` system module which they all import. The details of that module can be found in [26, Section 4] and need not concern us here. It is enough to say that they make precise the essential properties of configurations already discussed in Section 2, namely that they are multisets of objects and messages, that objects have the special syntax already discussed and each has a set of attribute-value pairs whose values are in a sort `Value`, etc. In addition, appropriate messages and rewrite rules to query the attributes of an object are also included in the `CONFIGURATION` module.

We assume that rewrite theories have an *order-sorted* type structure [14]; this means that they are typed (we call their types *sorts*), that types can have subtypes (which can be declared by *subsort* declarations) and that operation symbols (which are declared with the types of their arguments and the type of their result) can be overloaded. In addition to the order-sorted syntax, a rewrite theory declares the relevant structural laws so that rewriting can take place modulo those axioms, and of course the rewrite rules of the theory. In the `CONFIGURATION` module the operators $\_\_$ and $\_,\_$ for forming unions of configurations and of attribute-value pairs respectively have both been declared with structural laws of associativity, commutativity, and identity.

The translation of a given object-oriented module extends the basic structure of configurations with the classes, data sorts, messages and rules introduced by the module. In particular, this extension gives rise to a series of subsort declarations. All sorts originally declared as values of attributes are now included as subsorts of the `Value` sort. Similarly, all classes, in this case `BdBuff`, are declared as subsorts of the `Object` sort. Note that, in addition, a subsort `<BdBuff` of the sort `CId` of *class identifiers* has been introduced. The purpose of this subsort is to range over the class identifiers of the subclasses of `BdBuff`. For the moment, no such subclasses have been introduced; therefore, at present the only constant of sort `<BdBuff` is the class identifier `BdBuff`. Notice the slight ambiguity introduced by this notation, since now `BdBuff` denotes *two different things*: a *sort name* in the sort structure of a module, and a *data element* in a subsort of a data type of class identifiers. However, this ambiguity is harmless—the context will always make explicit the intended sense—and could in any case be easily avoided by an appropriate notational convention; for example, by adopting quotes for the identifier use.

Objects of sort `BdBuff` are defined by a predicate called a *sort constraint* and introduced by the keyword `sct` which they must satisfy. In this case the sort constraint requires that the class identifier must have sort `<BdBuff` and that it must have among its attributes an attribute called `contents` whose value must be a list of natural numbers, and attributes called `in` and `out` whose values must be natural numbers. For more on sort constraints see [27, 26].

A trivial observation that is however key to our solution of the inheritance anomaly is that

> If a variable $x$ is declared to have a sort $s$, then it can range over elements of that sort or of any of its subsorts.

This observation is used crucially in the above translation of the rewrite rules, so that the translated rules become fully *general* in the sense that they can apply not only to objects in the original class where they were defined, but also—as further explained in Section 4—to objects in its subclasses. For example, the rewrite rules originally introduced in the **BD-BUFF** module have been modified to make them applicable not only to objects whose class identifier is exactly **BdBuff**, but also to other objects with class identifiers for subclasses of **BdBuff**, which may in addition have other attributes, i.e., indeed to all the objects of the class **BdBuff**.

Specifically, whenever a class identifier `C` appears in the lefthand side of a rule declared in an object-oriented module, in its translation we understand that a variable ranging over `<C`—which can match the constant `C` and any other constants `C'` that could be introduced later in subsorts `<C'` of `<C` for subclasses `C'` of `C`—is meant instead.[8] In addition, in the translated form of the rules variables of the form **ATTS**, which match a set of additional attribute-value pairs, have been added to the patterns of objects. In this way the translated rules will also apply in subclasses where more attributes have been declared.

# 4 Class Inheritance

Our generalization of rewrite rules in the previous translation so that they can apply not only to objects in the original class but also to objects in any of its subclasses is motivated by a sharp distinction between two different notions of inheritance: *class inheritance* and *module inheritance*. At a type-theoretic level of data sorts and object classes, sort and class inheritance provides a means of *classifying* data and objects into taxonomic *hierarchies*. At the level of modules, module inheritance supports modularity, reuse, and ease of evolution by arranging modules into hierarchies and by providing a rich algebra of module compositionality operations.

Class inheritance is directly supported by the order-sorted type structure of rewriting logic. As we shall see in this section, a subclass declaration `C < C'` in an object-oriented module is just a particular case of a subsort declaration `C < C'`. As a consequence of the order-sorted type structure, the effect of a subclass declaration is that the attributes, messages and rules of all the superclasses as well as the newly defined attributes, messages and rules of the subclass characterize the structure and behavior of the objects in the subclass. An object in the subclass behaves exactly as any object in any of the superclasses, but it may exhibit additional behavior due to the introduction of new attributes, messages and rules in the subclass.

This notion of class inheritance is considerably more restrictive than, and should be sharply distinguished from, the notion of inheritance adopted in prac-

[8] This way of generalizing rules so that they can be inherited was pointed out in Section 4.4 of [24]; I am indebted to Timothy Winkler for later suggesting to me the elegant sort structure of the sorts `<C` as a better alternative to a more cumbersome identifier data type definition.

tice by most object-oriented languages, where the behavior of a message (sometimes also called a *method*) in a subclass may be different form its behavior in a superclass and where mechanisms to change message behavior by what is called message (or method) "specialization" or "derivation" are typically provided.

The need for such message specializations often appears in practice. However, in our approach the mechanisms for message specialization that change the previous behavior in a superclass belong to module inheritance and are cleanly separated for the more restrictive notion of class inheritance that we propose. The advantages of this separation are many; some will become apparent in this paper, others are discussed in [26], and still others will be the subject of a future paper.

For the moment we can point out that in this way we avoid doing violence to class inheritance by forcing upon it the job of modifying code. As a consequence, we can have great flexibility of code reuse *and* a precise and satisfactory *order-sorted* semantics for subclasses that respects the intuitions of what it means to *classify* objects. By contrast, in approaches that conflate these two equally laudable goals, flexibility of code reuse is achieved at the heavy price of emptying the notion of class of most of its conceptual value.

One important advantage of our notion of class inheritance is that *rewrite rules are always inherited downwards* in the class hierarchy. We can illustrate this point by defining a subclass of `BdBuff` with a new message `get2` to get two elements of the buffer at once. We can introduce such a subclass in the module

```
omod BD-BUFF2 is
  extending BD-BUFF .
  class BdBuff2 .
  subclass BdBuff2 < BdBuff .
  msg get2from_replyto_ : OId OId -> Msg .
  msg to_2elts-in_are_ : OId OId List -> Msg .
  vars B I : OId .
  vars E E' N M : Nat .
  var Q : List .
  rl (get2from B replyto I)
       < B : BdBuff2 | contents: Q E' E, in: N, out: M > =>
       < B : BdBuff2 | contents: Q, in: N, out: M + 2 >
       (to I 2elts-in B are E E') .
       ***the requirement of having at least two elements
       ***in the buffer is implicit in the pattern Q E' E
endom
```

The translation into a rewrite theory is given by the system module

```
mod BD-BUFF2# is
  extending BD-BUFF# .
  sorts <BdBuff2 BdBuff2 .
  subsort BdBuff2 < BdBuff .
  subsort <BdBuff2 < <BdBuff .
```

```
    op BdBuff2 : -> <BdBuff2 .
    op get2from_replyto_ : OId OId -> Msg .
    op to_2elts-in_are_ : OId OId List -> Msg .
    var Y : <BdBuff2 .
    vars B I : OId .
    vars E E' N M : Nat .
    var Q : List .
    var ATTS : Attributes .
    sct < B : Y | contents: Q, in: N, out: M, ATTS > : BdBuff2 .
    rl (get2from B replyto I)
         < B : Y | contents: Q E' E, in: N, out: M, ATTS > =>
         < B : Y | contents: Q, in: N, out: M + 2, ATTS >
         (to I 2elts-in B are E E') .
  endm
```

The consequence of this definition is that buffers in **BdBuff2** will react to **put** and **get** messages exactly like buffers in **BdBuff**. This is because the rewrite rules in **BD-BUFF#** have a variable **X** ranging over class identifiers in **<BdBuff**, and in the module **BD-BUFF2#** there is a subsort declaration

```
    subsort <BdBuff2 < <BdBuff .
```

Therefore, the variable **X** can match the constant **BdBuff2** of sort **<BdBuff2**, so that the rules in **BD-BUFF#** also apply to objects in **BdBuff2**.

The only difference between both classes is that, unlike buffers in **BdBuff**, buffers in **BdBuff2** can react to **get2** messages by sending the two rightmost elements if they exist. Note that, due to the fact that **Y** has sort **<BdBuff2**, the above rule will *not* match bounded buffers in **BdBuff** that are not in **BdBuff2**.

In the same vein, we could have defined additional messages, such as for example a message **empty?** that checks whether the buffer is empty or not, and could have introduced additional attributes, which could appear in the rules for those new messages without any problem.

An interesting example involving *multiple* class inheritance is that of a lockable bounded buffer. The idea is to have a class of lockable objects in general, and then define lockable bounded buffers by multiple inheritance from bounded buffers and from lockable objects. When the object is locked no messages except **unlock** should have any effect. Unlike the standard solution, which would add a Boolean-valued attribute to ascertain the locked or unlocked state of an object and would violate class inheritance in our sense, our solution is simpler and fully respects our notion of class inheritance. The standard solution could be achieved by means of module inheritance mechanisms to be discussed in Section 5.

The basic idea is to view the locking of an object as a kind of *metamorphosis* that changes the nature of the object. This suggests that the class of the object in fact *changes* when being locked. This can be easily accomplished by assuming that the sort **CId** of class identifiers has subsorts **UCId**, and **QCId** of unquoted and quoted identifiers together with quote and unquote operators

```
'_ : UCId -> QCId
unquote: QCId -> UCId
```

each inverse of the other, and by adopting the syntactic convention that the class identifiers introduced by users are always unquoted.[9] We can then define a module

```
omod LOCKABLE is
  class Lockable .
  msgs lock, unlock : OId OId -> Msg .
  var O : OId .
  var ATTS : Attributes .
  rl lock(O)
       < O : Lockable | ATTS > => < O : 'Lockable | ATTS > .
       ***the class changes from Lockable to 'Lockable
  rl unlock(O)
       < O : 'Lockable | ATTS > => < O : Lockable | ATTS > .
       ***the class changes from 'Lockable to Lockable
endom
```

whose corresponding translation into a rewrite theory is

```
mod LOCKABLE# is
  extending CONFIGURATION .
  sorts <Lockable Lockable .
  subsort Lockable < Object .
  subsort <Lockable < UCId .
  op Lockable : -> <Lockable .
  ops lock, unlock : OId -> Msg .
  var Z : <Lockable .
  vars O : OId .
  var ATTS : Attributes .
  sct < O : Z | ATTS > : Lockable .
  rl lock(O) < O : Z | ATTS > => < O : 'Z | ATTS > .
  rl unlock(O) < O : 'Z | ATTS > => < O : Z | ATTS > .
endm
```

Now we can define lockable bounded buffers by multiple inheritance as follows

```
omod LOCKABLE-BD-BUFF is
  extending LOCKABLE .
  extending BD-BUFF .
  class LckblBdBuff .
  subclasses LckblBdBuff < Lockable BdBuff .
endom
```

[9] Therefore, if this convention were to be followed, all the previous occurrences of the sort CId should be replaced by UCId.

We can pause for a moment and ask how rewriting logic and the notion of class inheritance that we have proposed contribute to solving the inheritance anomaly in the case where the behavior of messages in superclasses is not modified. An answer to this question can be summarized as follows:

1. Programming a class with rewrite rules completely eliminates any need for special code to enforce synchronization constraints. We only need to give rewrite rules specifying the desired behavior. The effect of synchronization is obtained *automatically* and *implicitly* by the very definition of deduction in the logic.
2. If a class `C` is a subclass of other previously defined classes in the precise sense that we have given to class inheritance in our framework, then all the rewrite rules defining messages in the superclasses are automatically inherited without any change whatsoever. Therefore, for cases of inheritance that fall within our precise technical notion of class inheritance, the inheritance anomaly *completely vanishes*.

We now need to consider cases where the inheritance anomaly appears in the context of message (or method) "specializations" that change the original behavior. In our framework those cases fall outside class inheritance and are dealt with by different mechanisms of module inheritance.

## 5 Module Inheritance and Message Specialization

In programming practice one often wants to *modify* the original code of an application to adapt it to a different situation. The class inheritance mechanism as we have defined it will *not* help in such cases: it is not its purpose, and forcing it to modify code would only muddle everything and destroy its semantics. Instead, what we propose is to provide different *module inheritance* mechanisms to do the job of code modification. This distinction between a type-theoretic level of classes (more generally sorts) and a level of modules which, in our case, are theories in rewriting logic was already clearly made in the FOOPS language (besides the original paper [12], see also [17] for a very good discussion of inheritance issues and of the class-module distinction in the context of FOOPS), and indeed goes back to the distinction between sorts and modules in OBJ [16].

In Maude, code in modules can be modified or adapted for new purposes by means of a variety of module operations—and combinations of several such operations in *module expressions*—whose overall effect is to provide a very flexible style of software reuse that can be summarized under the name of *module inheritance*. Module operations of this kind include:

1. *importing* a module in a `protecting`, `extending`, or `using` mode;
2. *adding new rewrite rules* to an imported module;
3. *renaming* some of the sorts or operations of a module;
4. *instantiating* a parameterized module;
5. *adding modules* to form their union;

6. *redefining* an operator—for example a message—so that its syntax and sort requirements are kept intact, but its semantics can be changed by discarding previously given rules involving the operator so that new rules or equations can then be given in their place;
7. *removing* a rule or an equational axiom, or removing an operator or a sort altogether along with the rules or axioms that depend on it so that it can be either discarded or replaced by another operator or sort with different syntax and semantics.

The operations 1–5 are all exactly as in OBJ3 [16]. The operations 6–7 are new and give a simple solution to the thorny problem of *message (or method) specialization* without complicating the class inheritance relation, which remains based on an order-sorted semantics. The need for message specialization, i.e., for providing a different behavior for a message, arises frequently in practice. Consider for example a message **gget** which behaves just as **get**, except that it cannot be accepted if the last message received was a **put**. This means that now bounded buffers must be *history sensitive*, that is, they must remember more about their past than was previously necessary. Specifically, for **gget** to behave correctly, not only must **put** leave somehow a trace of being the last message received, but any message other than **put** must, when accepted, erase such a trace. This of course requires redefining the messages in question.

Therefore, our solution is to understand this as a module inheritance problem, and to carefully distinguish it from class inheritance. In this case, it is the *modules* in which the classes are defined that stand in an inheritance relation, not the classes themselves. The *redefine* operation, with keyword **rdfn**, provides the appropriate way of modifying and inheriting the BD-BUFF module as shown below. To illustrate the differences between class and module inheritance, we define a module BD-BUFF+HS-BD-BUFF in which the old class of bounded buffers and the new, history-sensitive, class both coexist.

```
omod BD-BUFF+HS-BD-BUFF is
  protecting BOOL .
  extending BD-BUFF .
  using BD-BUFF*(class BdBuff to HSBdBuff, rdfn(msg put_in_,
        msg getfrom_replyto_)) .
  att after-put: Bool in HSBdBuff .
  initially contents: nil, in: 0, out: 0, after-put: false .
  msg ggetfrom_replyto_ : OId OId -> Msg .
  vars B I : OId .
  vars E N M : Nat .
  var Q : List .
  var Y : Bool .
  rl (put E in B)
    < B : HSBdBuff | contents: Q, in: N, out: M, after-put: Y >
    => < B : HSBdBuff | contents: E Q, in: N + 1, out: M,
          after-put: true > if (N - M) < bound .
      ***put acts as before, but after-put is set to true
```

```
   rl (getfrom B replyto I)
      < B : HSBdBuff | contents: Q E, in: N, out: M,
        after-put: Y > =>
      < B : HSBdBuff | contents: Q, in: N, out: M + 1,
        after-put: false > (to I elt-in B is E) .
        ***get acts as before, but after-put is set to false
   rl (ggetfrom B reply to I)
      < B : HSBdBuff | contents: Q E, in: N, out: M,
        after-put: false > =>
      < B : HSBdBuff | contents: Q, in: N, out: M + 1,
        after-put: false > (to I elt-in B is E) .
        ***gget acts like get, but only if after-put is false
endom
```

The module expression

```
using BD-BUFF*(class BdBuff to HSBdBuff, rdfn(msg put_in_,
     msg getfrom_replyto_)) .
```

declares that a new copy of the BD-BUFF module is created[10] and imported in such a way that the class BdBuff is renamed to HSBdBuff and the messages (msg put_in_) and (msg getfrom_replyto_) are both *redefined*, i.e., their syntax and sort information are maintained, but, within this new copy of the BD-BUFF module, the original rules defining their behavior are discarded. Their new behavior is then defined by the new rules for put and get given later in the module. In addition, the gget message is introduced and a rule defining its behavior is given. Notice that all the rules for objects in HSBdBuff use the newly defined attribute after-put, introduced by the statement

```
att after-put: Bool in HSBdBuff .
```

Space limitations preclude giving a detailed account of the rdfn and rmv (remove) commands; this will be done elsewhere.

The essential point to bear in mind about the module BD-BUFF+HS-BD-BUFF is that, although the classes BdBuff and HSBdBuff are both subsorts of Object, the class HSBdBuff is *not* a subclass of BdBuff. Therefore, in the context of the new module, the old rules for put and get messages and the new rules for put, get, and gget *coexist without interference*, because they apply to different objects in two different classes that are *incomparable* in the class hierarchy.

The distinction between class inheritance and module inheritance can be illustrated in this example by means of the diagrams in Figure 1, where the diagram on the left expresses the class inheritance relation between the three classes involved, and the diagram on the right expresses the module inheritance relation between the modules used to define those classes. Note that the arrows in the subclass relation have a very specific meaning, namely that of a

[10] However, submodules below BD-BUFF, such as the implicitly given CONFIGURATION module, are not copied: they are *shared*.

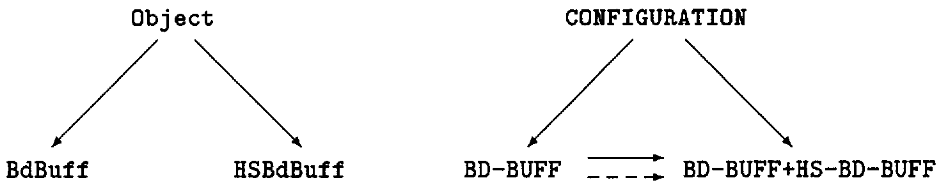


**Fig. 1.** Class inheritance vs. module inheritance for bounded buffers.

*subsort* relation, whereas the inheritance arrows between modules can have a much more flexible—yet precise—variety of meanings, because of the variety of module operations that can be involved. In this case, the solid arrows correspond to inheritance by **extending** importation, whereas the dotted arrow involves sort renaming, message redefinition, and a **using** importation; note also that the module BD-BUFF is inherited in *two* different ways by the module BD-BUFF+HS-BD-BUFF.

## 6 The Cheshire Cat

We can now attempt to summarize our discussions in previous sections and try to explain the nature and specific characteristics of the solution to the inheritance anomaly that we have proposed, and why it provides in our view a satisfactory solution to the problem.

The main goal, as already mentioned in the Introduction, is to solve the problem by making it disappear completely. In this sense, we should hope that the inheritance anomaly becomes like the Cheshire cat in Alice in Wonderland, which first disappears leaving its grin behind, and then the grin also disappears with the cat never coming back again.

In order to gain some feeling for how close we have been able to get to reaching such a goal, we should first point out that our treatment of the problem has in any case eliminated completely any need for special code for enforcing synchronization constraints. In fact, it seems to us that it is precisely the low level of abstraction involved in such synchronization constraints and the associated and necessary involvement of a language user under those circumstances into implementation decisions of which he should have been spared by a compiler and a higher level language that give rise to the "anomaly" in the first place. This does not imply in any way a lack of awareness of how useful and important it may be to give the advanced user of a concurrent language adequate ways of controlling and modifying the concurrent execution of his or her programs. However, there are disciplined ways of achieving such control by methods such as metaobject protocols [21] and other reflective methods [34].

Secondly, we should point out that—by introducing a clear distinction between class and module inheritance—our analysis has revealed that what is usually grouped together as a single problem can be fruitfully decomposed into two problems:

1. the case in which, as in the **get2** and the lockable bounded buffer examples discussed in Section 4, the behavior of messages previously defined in superclasses is not contradicted by their behavior in a subclass, and
2. the case in which, as in the **gget** example, the behavior of messages previously defined is in fact modified.

Regarding case 1, the semantics of class inheritance that we have proposed makes the inheritance anomaly problem disappear completely *without having to do anything*. Besides, the notion of class inheritance sheds light on when solutions based on the notion of a *guard* [23, 9], which have some similarities with rewrite rules, are likely to be most successful.

Case 2 involves an ineliminable need to *modify* the original behavior of some messages in ways that, in principle, would be hard if not impossible to foresee in advance. Therefore, *something* must necessarily be done. Our proposed solution is based on using module inheritance mechanisms that redefine the appropriate messages or remove some rules, and that create new classes with different behavior that are unrelated in the class inheritance hierarchy to the old classes that exhibited the original behavior. This solution has in our view the following advantages:

- it is fully general and gives complete flexibility for redefining the behavior of messages;
- it is achieved in a disciplined way by means of well-structured module operations which can be given a precise semantics as operations on logical theories;
- leaves intact the order-sorted semantics of class inheritance, and in fact operates outside the framework of class inheritance in our sense;
- *has nothing to do with concurrency*, and as before does not involve nor requires any special code for enforcing synchronization constraints.

In summary, the alleged incompatibility between inheritance and concurrency, the feeling that, somehow, it is problematic or very difficult to have both concurrency and inheritance coexisting in a satisfactory way within the same language, seems in ultimate analysis a mirage.

An area where fruitful research could be done is in devising more refined module inheritance mechanisms that reduce even more the work needed to modify the code in modules. As already pointed out, such techniques have nothing to do with concurrency and would in any case be very useful for many other languages, not necessarily object-oriented, and not necessarily concurrent.

A worthwhile research topic is transferring the ideas that have been developed here by means of rewriting logic techniques to other concurrent languages. This might cast new light on the strengths and limitations of existing inheritance techniques, and might suggest new language design solutions.

Yet another area where very useful research could be done is in devising efficient language implementation techniques that—when applied to languages supporting a high enough level of abstraction—avoid altogether the inheritance anomaly. Section 7 below further discusses this last topic.

# 7 Implementation Issues

It could perhaps be objected that, although rewriting logic and its order-sorted type structure together with the module inheritance mechanisms seem to make the inheritance anomaly go away, this has only been accomplished at the *specification level*, and that therefore we are after all somehow still left with the problem at the *implementation level.*

To answer this objection, we must clarify an implicit ambiguity between our use of rewriting logic as a language-independent semantic framework, and as a programming language with the Maude syntactic conventions. Indeed, in this paper rewriting logic has actually been used for *both* purposes. However, it would not be reasonable to implement rewriting logic in its fullest generality for programming purposes. This is because, in its most general form, rewriting can take place *modulo* an arbitrary equational theory $E$ that could be undecidable. Therefore, for programming purposes rewriting logic must be carefully restricted in order to allow reasonably efficient implementations. We discuss below specific restrictions under which parallel implementations could be developed. Therefore, although our ideas are still preliminary, we intend our solution of the inheritance anomaly to work at both the specification and the implementation levels.

We consider two subsets of rewriting logic. The first subset allows rewriting modulo any combination of a few commonly occurring structural axioms such as associativity, commutativity and identity for which matching algorithms exist. This subset gives rise to Maude—in the sense that Maude modules are executable rewriting logic theories in it—and, although it can in some cases be inefficient, can be supported by an interpreter implementation adequate for rapid prototyping, debugging, and executable specification. The second, smaller subset gives rise to Simple Maude, a sublanguage meant to be used for concurrent programming purposes for which a wide variety of machine implementations can be developed. Figure 2 summarizes the three levels involved.

## 7.1 Simple Maude as a Machine-Independent Parallel Language

Simple Maude represents our present design decisions about the subset of rewriting logic that could be implemented efficiently in a wide variety of machine architectures. In fact, we regard Simple Maude as a *machine-independent parallel programming language*, which could be executed with reasonable efficiency on many parallel architectures.

Communication in Simple Maude is performed by asynchronous message passing. The restriction from Maude to Simple Maude is obtained by restricting the form of the rewrite rules that are allowed. We refer the reader to [30] for more

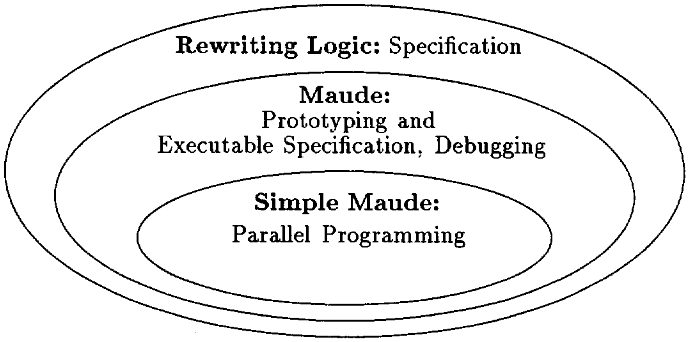


**Fig. 2.** Maude and Simple Maude as subsets of Rewriting Logic.

details about Simple Maude and concentrate only on the case of object-oriented modules, where we only allow conditional rules of the form

$$
\begin{aligned}
(\ddagger) \quad & (M)\ \langle O : F \mid \mathit{atts} \rangle \\
& \longrightarrow (\langle O : F' \mid \mathit{atts}' \rangle) \\
& \quad \langle Q_1 : D_1 \mid \mathit{atts}''_1 \rangle \ldots \langle Q_p : D_p \mid \mathit{atts}''_p \rangle \\
& \quad M'_1 \ldots M'_q \\
& \quad \mathit{if}\ C
\end{aligned}
$$

involving only one object and one message in their lefthand side, where $p, q \geq 0$, and where the notation $(M)$ means that the message $M$ is only an optional part of the lefthand side, that is, that we also allow *autonomous objects* that can act on their own without receiving any messages. Similarly, the notation $(\langle O : F' \mid \mathit{atts}' \rangle)$ means that the object $O$—in a possibly different state—is only an optional part of the righthand side, i.e., that it can be omitted in some rules.

Specifically, the lefthand sides in rules of the form ($\ddagger$) should fit the general pattern

$$M(O)\ \langle O : C \mid \mathit{atts} \rangle$$

where $O$ could be a variable, a constant, or more generally—in case object identifiers are endowed with additional structure—a term. Under such circumstances, an efficient way of realizing rewriting modulo associativity and commutativity by communication is available to us for rules of the form ($\ddagger$), namely we can associate object identifiers with specific addresses in the machine where the object is located and send messages addressed to the object to the corresponding address.

A declarative version of the Actor model [2, 1] can be obtained by only allowing rules of the form ($\ddagger$) with the additional restrictions of necessarily involving a message in the lefthand side and of being *unconditional* (see Section 4.6 of [26] for a discussion of the actor model and its rewriting logic semantics). Therefore, in spite of the restrictions imposed on it, Simple Maude is still quite expressive and is in particular more expressive than actors.

### 7.2 MIMD, SIMD, and MIMD/SIMD Implementations

Although we are still in the planning stages of language implementation, the ample experience that already exists on efficient compilation of rewriting for functional languages, and our past experience on parallel compilation of rewrite rules for the Rewrite Rule Machine [4] lead us to believe that Simple Maude can be implemented with reasonable efficiency on a wide variety of parallel architectures, including MIMD, SIMD, and MIMD/SIMD architectures.

Each of these architectures is naturally suited for a different way of performing rewriting computations. Simple Maude has been chosen so that concurrent rewriting with rules in this sublanguage should be relatively easy to implement in any of these three classes of machines. The paper [30] discusses this matter in greater detail; here we limit ourselves to a brief sketch.

In the MIMD (multiple instruction stream, multiple data) case many different rewrite rules can be applied at many different places at once, but only one rule is applied at one place in each processor. The implementation of object-oriented rules of the form (‡), involving a message and an object, can be achieved by *interprocessor communication*, sending the message to the processor in which the addressee object is located, so that when the message arrives the corresponding rules can be applied.

The SIMD (single instruction stream, multiple data) case corresponds to applying rewrite rules one at a time, possibly to many places in the data. The implementation of rules of the form (‡) will require special SIMD code for message passing in addition to the SIMD code for performing the rewriting.

The MIMD/SIMD case is at present more exotic; the Rewrite Rule Machine (RRM) [15, 5, 4, 3] is an architecture in this class in which the processing nodes are two-dimensional SIMD arrays realized on a chip and the higher level structure is a network operating in MIMD mode. This case corresponds to applying many rules to many different places in the data, but here a single rule may be applied at many places simultaneously within a single processing node. The message passing required for rules of the form (‡) can be performed in a way entirely similar to the MIMD case. From the point of view of maximizing the amount and flexibility of the rewriting that can happen in parallel, the MIMD/SIMD case provides the most general solution and offers the best prospects for reaching extremely high performance in many applications.

## 8 Concluding Remarks

This paper has presented a solution to the inheritance anomaly based on rewriting logic that eliminates the need for explicit synchronization code and removes any obstacles to the full integration of concurrency and inheritance within a concurrent object-oriented language. This work suggests further work ahead on language design for concurrent object-oriented languages and on efficient implementation techniques supporting the level of abstraction desirable for concurrent languages that aim at avoiding altogether the inheritance anomaly.

In particular, it might be fruitful to investigate how the ideas presented here could be used in other languages whose syntax may be quite different from that of rewrite rules. The semantic framework of rewriting logic is very general, can support both synchronous and asynchronous communication, and could probably be usefully applied to many of those languages precisely for this purpose. The work presented here offers a solution from a particular perspective, namely one that views programs as collections of rewrite rules. In order to exploit the techniques available from this perspective in the context of other concurrent object-oriented languages more research clearly needs to be done.

Of the previous proposals in the literature, the closest in spirit to the present one are those by Matsuoka and Yonezawa [23] and by Frolund [9]. We comment briefly on their relationships to the present work. Both Matsuoka and Yonezawa and Frolund advocate *guards* as a useful and fairly reusable way of writing synchronization code. Although, as already pointed out, our approach completely eliminates the need for any special synchronization code, the *implicit effect* of guards is obtained by the patterns in rewrite rules and by the conditions in conditional rewrite rules and therefore there is some similarity between those two approaches and ours. Roughly speaking, our notion of class inheritance identifies a type of inheritance situation where guards can work very well.

The relationship with the work of Frolund [9] could be summarized by saying that, although Frolund allows more general cases of (class) inheritance than we do—so that some of his examples would in our case be treated by means of module inheritance techniques—however, his approach is somewhat more restrictive in the sense that, for safety reasons, he adopts the position that a method's synchronization constraints should increase monotonically as we go down in the inheritance hierarchy. By contrast, when defining a subclass with our treatment of class inheritance, we can not only add new rules for new messages, but we can also add new rules for messages previously defined in some superclasses, and this can have the effect of extending the behavior of those previously defined messages. In terms of guards, this would correspond to relaxing for a subclass the conditions under which a message can be invoked, whereas in Frolund's treatment those conditions should become more restrictive.

Yet another point of similarity is Matsuoka's and Yonezawa's [23] goal of reducing synchronization code to the minimum, a goal fully consistent with the complete elimination of such code that we advocate. Finally, a different solution proposed by Matsuoka and Yonezawa [23] based on the use of *reflection* bears some resemblance to our use of module inheritance for cases where the behavior of messages has to be modified. The point is that, very roughly speaking, one could regard the module inheritance techniques that we have proposed as a very well structured form of static reflection where the code is modified at compile time.

**Acknowledgements**

I thank Akinori Yonezawa and Satoshi Matsuoka for kindly explaining to me the difficulties involved in the "inheritance anomaly" and their solutions to those difficulties along several very fruitful and illuminating conversations that, along with the reading of their clear and insightful paper [23], have stimulated my work on this topic. I also thank Satoshi Matsuoka and Svend Frolund for their very helpful comments to a previous version that have suggested improvements and clarifications in the exposition.

I thank my fellow members of the Declarative Languages and Architecture Group at SRI International, especially Timothy Winkler, Narciso Martí-Oliet and Patrick Lincoln, for the many discussions with them on object-oriented matters, and for their technical contributions to Maude that have benefited this work. In addition, Narciso Martí-Oliet deserves special thanks for his very helpful suggestions after carefully reading the manuscript.

I thank Joseph Goguen for our long term collaboration on the OBJ, Eqlog and FOOPS languages [16, 11, 12], concurrent rewriting [10] and its implementation on the RRM architecture [13, 4], all of which have influenced this work, and Ugo Montanari for our joint work on the semantics of Petri nets [28, 29] that was an important early influence on rewriting logic.

## References

1. G. Agha. *Actors.* MIT Press, 1986.
2. G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming,* pages 37–53. MIT Press, 1988.
3. H. Aida, J. Goguen, S. Leinwand, P. Lincoln, J. Meseguer, B. Taheri, and T. Winkler. Simulation and performance estimation for the rewrite rule machine. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation,* pages 336–344. IEEE, 1992.
4. Hitoshi Aida, Joseph Goguen, and José Meseguer. Compiling concurrent rewriting onto the rewrite rule machine. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems, Montreal, Canada, June 1990,* pages 320–332. Springer LNCS 516, 1991.
5. Hitoshi Aida, Sany Leinwand, and José Meseguer. Architectural design of the rewrite rule machine ensemble. In J. Delgado-Frias and W.R. Moore, editors, *VLSI for Artificial Intelligence and Neural Networks,* pages 11–22. Plenum Publ. Co., 1991. Proceedings of an International Workshop held in Oxford, England, September 1990.
6. Pierre America. Synchronizing actions. In *Proc. ECOOP'87,* pages 234–242. Springer LNCS 276, 1987.
7. Denis Caromel. Concurrency and reusability: from sequential to parallel. *Journal of Object-Oriented Programming,* pages 34–42, September/October 1990.
8. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B,* pages 243–320. North-Holland, 1990.

9. Sven Frolund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In O. Lehrmann Madsen, editor, *Proc. ECOOP'92*, pages 185–196. Springer LNCS 615, 1992.
10. Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In R. Keller and J. Fasel, editors, *Proc. Workshop on Graph Reduction, Santa Fe, New Mexico*, pages 53–93. Springer LNCS 279, 1987.
11. Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179-210, September 1984.
12. Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986.
13. Joseph Goguen and José Meseguer. Software for the rewrite rule machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 628–637. ICOT, 1988.
14. Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
15. Joseph Goguen, José Meseguer, Sany Leinwand, Timothy Winkler, and Hitoshi Aida. The rewrite rule machine. Technical Report SRI-CSL-89-6, SRI International, Computer Science Laboratory, March 1989.
16. Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1992. To appear in J.A. Goguen, editor, *Applications of Algebraic Specification Using OBJ*, Cambridge University Press.
17. Joseph Goguen and David Wolfram. On types and FOOPS. To appear in *Proc. IFIP Working Group 2.6 Working Conference on Database Semantics: Object-Oriented Databases: Analysis, Design and Construction, 1990.*
18. Gerard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27:797–821, 1980.
19. Yutaka Ishikawa. Communication mechanism on autonomous objects. In *OOPSLA'92 Conference on Object-Oriented Programming*, pages 303–314. ACM, 1992.
20. Dennis Kafura and Keung Lee. Inheritance in actor based concurrent object oriented languages. In *Proc. ECOOP'89*, pages 131–145. Cambridge University Press, 1989.
21. Gregor Kiczales, Jim des Riviers, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.
22. Henry Liebermann. Concurrent object-oriented programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1988.
23. Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. Dept. of Information Science, University of Tokyo, January 1991; to appear in G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, MIT Press, 1993.

24. José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.
25. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
26. José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. To appear in G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, MIT Press, 1993.
27. José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, 1993.
28. José Meseguer and Ugo Montanari. Petri nets are monoids: A new algebraic foundation for net theory. In *Proc. LICS'88*, pages 155–164. IEEE, 1988.
29. José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.
30. José Meseguer and Timothy Winkler. Parallel Programming in Maude. In J.-P. Banâtre and D. Le Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992.
31. Christian Neusius. Synchronizing actions. In Pierre America, editor, *Proc. ECOOP'91*, pages 118–132. Springer LNCS 512, 1991.
32. M. Papathomas. Concurrency issues in object-oriented programming languages. In D. Tsichritzis, editor, *Object Oriented Development*, pages 207–246. Université de Geneve, 1989.
33. Etsuya Shibayama. Reuse of concurrent object descriptions. In *Proc. TOOLS 3, Sydney*, pages 254–266, 1990.
34. Brian Smith and Akinori Yonezawa, editors. *Proc. of the IMSA'92 Workshop on Reflection and Meta-Level Architecture, Tama-city, Tokyo.* Research Institute of Software Engineering, 1992.
35. Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled sets. In *OOPSLA'89 Conference on Object-Oriented Programming*, pages 103–112. ACM, 1989.
36. Ken Wakita and Akinori Yonezawa. Linguistic support for development of distributed organizational information systems. In *Proc. ACM COCS*. ACM, 1991.
37. A. Yonezawa, J.-P. Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA'86 Conference on Object-Oriented Programming, Portland, Oregon, September-October 1986*, pages 258–268. ACM, 1986.


# A Rewriting Logic

This appendix gives the rules of deduction of rewriting logic.

## A.1 Basic Universal Algebra

For the sake of simplifying the exposition, we treat the *unsorted* case; the many-sorted and order-sorted cases can be given a similar treatment. Therefore, a set $\Sigma$ of function symbols is a ranked alphabet $\Sigma = \{\Sigma_n \mid n \in \mathbb{N}\}$. A $\Sigma$-algebra is then a set $A$ together with an assignment of a function $f_A : A^n \longrightarrow A$ for each $f \in \Sigma_n$ with $n \in \mathbb{N}$. We denote by $T_\Sigma$ the $\Sigma$-algebra of ground $\Sigma$-terms, and by $T_\Sigma(X)$ the $\Sigma$-algebra of $\Sigma$-terms with variables in a set $X$. Similarly, given a set $E$ of $\Sigma$-equations, $T_{\Sigma,E}$ denotes the $\Sigma$-algebra of equivalence classes of ground $\Sigma$-terms modulo the equations $E$ (i.e., modulo provable equality using the equations $E$); in the same way, $T_{\Sigma,E}(X)$ denotes the $\Sigma$-algebra of equivalence classes of $\Sigma$-terms with variables in $X$ modulo the equations $E$. Let $[t]_E$ or just $[t]$ denote the $E$-equivalence class of $t$.

Given a term $t \in T_\Sigma(\{x_1,\ldots,x_n\})$, and terms $u_1,\ldots,u_n$, $t(u_1/x_1,\ldots,u_n/x_n)$ denotes the term obtained from $t$ by *simultaneously substituting* $u_i$ for $x_i$, $i = 1,\ldots,n$. To simplify notation, we denote a sequence of objects $a_1,\ldots,a_n$ by $\overline{a}$. With this notation, $t(u_1/x_1,\ldots,u_n/x_n)$ can be abbreviated to $t(\overline{u}/\overline{x})$.

## A.2 The Rules of Rewriting Logic

A *signature* in rewriting logic is a pair $(\Sigma, E)$ with $\Sigma$ a ranked alphabet of function symbols and $E$ a set of $\Sigma$-equations. Rewriting will operate on equivalence classes of terms modulo the set of equations $E$. In this way, we free rewriting from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set $E$ of equations is empty. The idea of rewriting in equivalence classes is well known [18, 8].

Given a signature $(\Sigma, E)$, *sentences* of the logic are sequents of the form $[t]_E \longrightarrow [t']_E$ with $t, t'$ $\Sigma$-terms, where $t$ and $t'$ may possibly involve some variables from the countably infinite set $X = \{x_1,\ldots,x_n,\ldots\}$. A *theory* in this logic, called a rewrite theory, is a slight generalization of the usual notion of theory—which is typically defined as a pair consisting of a signature and a set of sentences for it—in that, in addition, we allow rules to be labelled. This is very natural for many applications, and customary for automata—viewed as labelled transition systems—and for Petri nets, which are both particular instances of our definition.

**Definition 1.** A (*labelled*) *rewrite theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where $\Sigma$ is a ranked alphabet of function symbols, $E$ is a set of $\Sigma$-equations, $L$ is a set of *labels*, and $R$ is a set of pairs $R \subseteq L \times (T_{\Sigma,E}(X)^2)$ whose first component is a label and whose second component is a pair of $E$-equivalence classes of terms, with $X = \{x_1,\ldots,x_n,\ldots\}$ a countably infinite set of variables. Elements of $R$ are called *rewrite rules*.[11] We understand a rule $(r, ([t],[t']))$ as a labelled

[11] To simplify the exposition the rules of the logic are given for the case of *unconditional* rewrite rules. However, all the ideas and results presented here have been extended to conditional rules in [25] with very general rules of the form

$$r : [t] \longrightarrow [t'] \ \textit{ if } \ [u_1] \longrightarrow [v_1] \wedge \ldots \wedge [u_k] \longrightarrow [v_k].$$

This of course increases considerably the expressive power of rewrite theories, as illustrated by several of the examples presented in this paper.

sequent and use for it the notation $r : [t] \longrightarrow [t']$. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either $t$ or $t'$, we write $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, or in abbreviated notation $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$.

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sequent $[t] \longrightarrow [t']$ and write $\mathcal{R} \vdash [t] \longrightarrow [t']$ if and only if $[t] \longrightarrow [t']$ can be obtained by finite application of the following *rules of deduction*:

1. **Reflexivity**. For each $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}$$

2. **Congruence**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t_1'] \quad \ldots \quad [t_n] \longrightarrow [t_n']}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t_1', \ldots, t_n')]}$$

3. **Replacement**. For each rewrite rule $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in $R$,

$$\frac{[w_1] \longrightarrow [w_1'] \quad \ldots \quad [w_n] \longrightarrow [w_n']}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}$$

4. **Transitivity**.

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

A nice consequence of having defined rewriting logic is that concurrent rewriting, rather than emerging as an operational notion, actually *coincides* with deduction in such a logic.

**Definition 2.** Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, a $(\Sigma, E)$-sequent $[t] \longrightarrow [t']$ is called a *concurrent $\mathcal{R}$-rewrite* (or just a *rewrite*) iff it can be derived from $\mathcal{R}$ by finite application of the rules 1-4.