To appear in G. Agha, P. Wegner, and A. Yonezawa (editors), *Research Directions in Object-Based Concurrency*, MIT Press, 1993.

A Logical Theory of Concurrent Objects and its Realization in the Maude Language*

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

Abstract

A new theory of concurrent objects is presented. The theory has the important advantage of being based directly on a simple logic called *rewriting logic* in which concurrent object-oriented computation exactly corresponds to logical deduction. This deduction is performed by *concurrent rewriting* modulo structural axioms of associativity, commutativity and identity that capture abstractly the essential aspects of communication in a distributed object-oriented configuration made up of concurrent objects and messages. This axiomatization of objects, classes, and concurrent object-oriented computations in terms of rewriting logic is proposed as a general semantic framework for object-oriented programming. A direct fruit of this theory is a new language, called Maude, that can be used to program concurrent object-oriented systems in an entirely declarative way using rewriting logic. Modules written in this language are used to illustrate the main ideas with examples. Maude supports a highly modular and parameterized programming style, contains OBJ3 as a functional sublanguage, and provides a simple and semantically rigorous unification of functional programming and concurrent object-oriented programming. A sublanguage called Simple Maude that can be implemented with reasonable efficiency on a wide variety of parallel architectures is described. The relationship with Actors and with other models of concurrent computation is discussed. An extension of Maude called MaudeLog is sketched; MaudeLog is also based on rewriting logic and unifies the paradigms of functional, relational, and concurrent object-oriented programming. The model theory of rewriting logic and an initial model semantics for Maude modules are also discussed.

Contents

*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).

1 Introduction

An important reason that makes object-oriented programming very attractive as a programming language paradigm is its conceptual support for structuring programs as systems made up of objects that interact with each other. Object-oriented concepts fit well with our intuitive ideas about ordinary objects and their interactions in the world and allow us to conceive of a program as either a *simulation* or model of some aspects of the world, or—when actual interaction with the world is desired, as for example in a robotics application—as the addition of a subsystem of *artificial objects* as artifacts that interact with other objects in the world.

Since interactions between objects in the real world are concurrent, it would seem to follow that concurrency should be viewed as an *intrinsic* property of object-oriented systems. Although this idea was certainly present in the Simula 67 language [32, 16] and was partially realized there within the constraints of a sequential implementation by means of the notion of "quasi-parallel" execution, the notion that concurrency is intrinsic to object-oriented programming seems to have been deemphasized or forgotten to a considerable extent in the subsequent evolution of the field, as the existence of the term "concurrent object-oriented programming" to indicate a delimited subfield concerned with "adding concurrency" to object-oriented programming seems to indicate.

This is an unsatisfactory state of affairs, especially when the following considerations are added:

1. At present it is considered very difficult to have both concurrency and inheritance in an object-oriented language. This difficulty is referred to as the "inheritance anomaly" [67, 90, 38].

2. Type-theoretic approaches to object-oriented programming which are sometimes put forward as providing a semantics for the field seem at present to be altogether silent on concurrency issues.

3. There seems to exist no agreement on a semantic basis for concurrent object-oriented programming. Wildly varying proposals to graft all sorts of concurrency constructs and models quite alien to the concepts of object-oriented programming into existing or new languages in an ad-hoc way are not only made, but are in some cases followed up with actual implementations. Certainly, Actors [3, 2] seems the best proposal so far, since actors do not suffer from the incoherence of other approaches and message passing is a very flexible communication mechanism, but the addition of inheritance to actor languages still seems to run into the difficulties already mentioned above.

4. Since just adding concurrency is at present somewhat of a stumbling block, it is not surprising that there is even less of an agreement on how concurrent object-oriented programming could be unified in a multiparadigm fashion with functional programming and with relational programming in a coherent and semantically rigorous way.

Regarding the last consideration, there are very good reasons for seeking a unification of these three paradigms or "perspectives" as Kristen Nygaard likes to call them. As Nygaard himself points out [89],

> "It seems obvious to the author that all these three perspectives should be supported within any new general programming language in the future. No perspective will "win" as some seem to believe."

Indeed, there is much to be gained in a unification of this kind if it is done right. By "done right" I mean that the overall unification should in fact be based directly on an adequate logic so that programs become theories in that logic and computation becomes logical deduction.

This, however, is not an easy task. The main difficulty has to do with logic, because the logics on which functional and relational programming are based—namely, equational logic (in either a first-order or a higher-order version) and first-order Horn logic, respectively—describe unchanging Platonic structures such as sets, functions and relations as it is fitting for logics originally introduced to develop logical foundations for mathematics. Such logics, however, deal very poorly with action and change, and deal particularly poorly with the type of change typical of concurrent object-oriented systems. This has many manifestations in practice, including among others the long-term embarrassment of the frame problem[1] in artificial intelligence.

This work proposes a simple logic of action called rewriting logic [70, 69, 72] as a general semantic framework for object-oriented programming with the following characteristics:

1. Concurrency is intrinsic and therefore the semantic framework formalizes *concurrent* object-oriented programming. However, concurrency is *implicit* and does not require any special extralogical constructs; this agrees with the view that concurrency is inherent to object-oriented systems and should therefore be directly supported by an adequate semantic framework.

2. A rigorous semantics for multiple class inheritance is provided in such a way that the so-called inheritance anomaly blocking the integration of concurrency and inheritance disappears completely.

3. A declarative version of the actor model appears as a special case, and is given a new logical and truly concurrent semantics based on rewriting logic.

4. Rewriting logic provides not only a semantic framework, but also a *computational model* for concurrent object-oriented programming. This allows a fully declarative programming style for concurrent object-oriented systems that can be programmed as theories in rewriting logic and whose concurrent computations correspond to logical deductions in such a logic. All this is realized in a language called Maude that is directly based on rewriting logic, and makes possible a natural integration of specification, programming, and formal reasoning within such a language.

5. Maude naturally unifies the functional programming paradigm with concurrent object-oriented programming and contains (a slight linguistic variant of) the OBJ language [45, 53] as its functional sublanguage. An extension of Maude called MaudeLog [73]—also based entirely on rewriting logic—unifies the three paradigms of functional, relational, and concurrent object-oriented programming.

6. Rewriting logic is sound and complete and has initial models. The mathematical semantics of Maude modules is based on such initial models which intuitively correspond to concurrent systems[2].

7. Like OBJ [53], Eqlog [48], and FOOPS [49], Maude has modularity and parameterization mechanisms à la Clear [20]. This, together with its support for multiple class inheritance and with two new module operations proposed in this paper, endows Maude with powerful mechanisms for program reuse and for programming-in-the-large by adapting and composing modules together in very flexible ways.

[1]For a detailed discussion of why rewriting logic entirely avoids the frame problem see [63].

[2]However, the denotational semantics of *functional* modules is given by the usual initial algebra semantics as in OBJ [51].

8. A sublanguage of Maude called Simple Maude [83] provides a machine independent parallel programming language that can be implemented with reasonable efficiency on a wide variety of parallel architectures. In addition, Simple Maude can be used to incorporate modules written in conventional code into parallel programs, and to integrate open heterogeneous systems in a parallel computing environment.

The paper discusses in more detail all the aspects just mentioned about rewriting logic as a semantic framework for concurrent object-oriented programming and about the Maude language, and illustrates many of the ideas with examples. Section 2 introduces Maude's functional and system modules and their concurrent rewriting computation, and discusses some basic order-sorted algebra concepts used throughout the paper. Section 3 introduces rewriting logic, identifies concurrent rewriting computation with deduction in such a logic, and discusses the intended meaning of the logic. Section 4 presents a logical theory of concurrent objects based on rewriting logic, introduces Maude's object-oriented modules, and discusses class and module inheritance, object creation and deletion, message broadcasting, reflection, and actors. A brief discussion of the generality of rewriting logic as a model of concurrency is also included at the end of the section. Section 5 introduces Simple Maude, summarizes implementation ideas for several parallel architectures, and briefly discusses support for multilingual extensions and open heterogeneous systems. Section 6 presents and discusses two longer Maude examples. Section 7 gives a brief sketch of Maude and MaudeLog as multiparadigm logic programming languages and discusses the multiparadigm unification of functional and concurrent object-oriented programming, and of functional, relational, and concurrent object-oriented programming that they respectively provide. Section 8 discusses the model theory of rewriting logic, including initial and free models, gives a computational interpretation of such models as concurrent systems, and defines the mathematical semantics of Maude modules in terms of initial models. Section 9 discusses related work. The paper ends with some concluding remarks in Section 10.

2 Maude and Concurrent Rewriting

Concurrent rewriting is motivated with examples of *functional* and *system* modules in Maude. The system module examples show that the traditional interpretation of rewrite rules as equations must be abandoned and that a new logic and model theory are needed. Rewriting logic—introduced in Section 3—provides the answer; in it, concurrent computation by rewriting coincides with logical deduction. Discussion of object-oriented aspects, and in particular of Maude's *object-oriented* modules, is deferred to Section 4.

2.1 Functional Modules

The idea of concurrent rewriting is very simple. It is the idea of *equational simplification* that we are all familiar with from our secondary school days, *plus* the obvious remark that we can do many of those simplifications independently, i.e., in *parallel.* Consider for example the following Maude functional modules written in a notation entirely similar to that of OBJ3 [45, 53]:

```
fmod NAT is
  sorts Nat NzNat .
  subsort NzNat < Nat .
  op 0 : -> Nat .
```

```
fmod NAT-REVERSE is
  protecting NAT .
  sort Tree .
  subsort Nat < Tree .
```

```
    op s_ : Nat -> NzNat .
    op p_ : NzNat -> Nat .
    op _+_ : Nat Nat -> Nat [comm] .
    vars N M : Nat .
    eq p s N = N .
    eq N + 0 = N .
    eq (s N) + (s M) = s s (N + M) .
  endfm
```

```
    op _^_ : Tree Tree -> Tree .
    op rev : Tree -> Tree .
    var N : Nat .
    vars T T' : Tree .
    eq rev(N) = N .
    eq rev(T ^ T') =
         rev(T') ^ rev(T) .
  endfm
```

The first module defines the natural numbers in Peano notation with successor, predecessor and addition functions, and the second defines a function to reverse a binary tree whose leaves are natural numbers. Each module begins with the keyword `fmod` followed by the module's name, and ends with the keyword `endfm`. A module contains sort and subsort declarations introduced by the keywords `sort(s)` and `subsort(s)` stating the different sorts of data manipulated by the module and how those sorts are related. As in OBJ3, Maude's type structure is *order-sorted* [51]; therefore, sorts form a partially ordered set and it is possible to declare one sort as a *subsort* of another; for example, the declaration `NzNat < Nat` states that every nonzero natural number is a natural number, and the declaration `Nat < Tree` states that every natural number is a tree consisting of a single node. It is also possible to *overload* function symbols for operations that are defined at several levels of a sort hierarchy and agree on their results when restricted to common subsorts; for example, an addition operation `_+_` may be defined for sorts `Nat`, `Int`, and `Rat` of natural, integer, and rational numbers with

```
  Nat < Int < Rat .
```

Each of the functions provided by the module, as well as the sorts of their arguments and the sort of their result, is introduced using the keyword `op`. The syntax is user-definable, and permits specifying function symbols in "prefix," (in the `NAT` example the functions `s_` and `p_`), "infix" (`_+_`) or any "mixfix" combination as well as standard parenthesized notation (`rev`).

A *functional* model for such an order-sorted syntax is called an *order-sorted algebra* [51] and consists of a set A_s for each sort symbol s, so that if $s \leq s'$ then $A_s \subseteq A_{s'}$, together with a function $f_A : A_{s_1} \times \ldots \times A_{s_n} \longrightarrow A_s$ for each operator declaration $f : s_1 \ldots s_n \longrightarrow s$ in such a way that if another operator declaration $f : s'_1 \ldots s'_n \longrightarrow s'$ has been given, with $s'_i \leq s_i$, $i = 1, \ldots, n$, and $s' \leq s$, then the function $f_A : A_{s'_1} \times \ldots \times A_{s'_n} \longrightarrow A_{s'}$ is just the restriction of the function $f_A : A_{s_1} \times \ldots \times A_{s_n} \longrightarrow A_s$ to the subset $A_{s'_1} \times \ldots \times A_{s'_n}$. The number hierarchy from the naturals to the rationals (and of course beyond) provides a good example of an order-sorted algebra.

Variables to be used for defining equations are declared with their corresponding sorts, and then equations are given; such equations provide the actual "code" of the module. Deduction with such equations is a typed variant of equational logic called *order-sorted equational logic*. However, operationally, only deduction from left to right by rewriting is performed, as explained below. As in OBJ3, the *mathematical semantics* of a Maude functional module is the *initial* order-sorted algebra [51] satisfying the equations declared in the module. For the two modules above, such initial algebras are just what we would expect, namely the natural numbers—with zero, successor, predecessor and addition—for the `NAT` module, and binary trees with natural numbers on their leaves—with tree reversal and binary tree constructor operators—for the figure=/homes/dumas/winkler/text/fig/p.ps,scale=100

Figure 1: Concurrent rewriting of a tree of numbers.

NAT-REVERSE module, in which the natural numbers are viewed as the subset of trees consisting of a single node.

The statement protecting NAT imports NAT as a *submodule* of NAT-REVERSE and asserts that the natural numbers are not modified in the sense that no new data of sort Nat is added, and different numbers are not identified by the new equations declared in NAT-REVERSE.

To compute with such modules, one performs equational simplification by using the equations from left to right until no more simplifications are possible. Note that this can be done *concurrently*, i.e., applying several equations at once, as in the example of Figure 1, in which the places where the equations have been matched at each step are marked. Notice that the function symbol _+_ was declared to be commutative by the attribute[3] [comm]. This not only asserts that the equation N + M = M + N is satisfied in the intended semantics, but it also means that when doing simplification we are allowed to apply the rules for addition not just to *terms*—in a purely syntactic way—but to *equivalence classes* of terms *modulo* the commutativity equation. In the example of Figure 1, the equation N + 0 = N is applied (modulo commutativity) with 0 both on the right *and* on the left.

A particularly appealing feature of this style of concurrent programming is the *implicit* nature of the parallelism. Since in the two modules above the equations are *Church-Rosser* (also called *confluent*) and *terminating* (see Section 3.3 for a definition of these notions, [56, 33] for further background on the subject, and [59] for corresponding order-sorted versions of such notions) the *order* in which the rules are applied does not at all affect the final result. This agrees with the usual functional programming expectations, since it would be quite strange to obtain different results for the same functional expression. Indeed, in Maude, the rules in a *functional* module are always assumed to be Church-Rosser, but as we shall see later this is not assumed for either system modules or object-oriented modules, which do *not* have a functional interpretation.

2.1.1 Parameterized Functional Modules

As in OBJ3, functional modules can be *parameterized*. For example, we can define a parameterized module by generalizing the NAT-REVERSE module to a parameterized REVERSE[X :: TRIV] module in which the set of data that can be stored in tree leaves is a parameter. In parameterized modules, the properties that the parameter must satisfy are specified by one or more *parameter theories*. In this case, the (functional) parameter theory is the trivial theory TRIV

```
fth TRIV is
  sort Elt .
endft
```

which only requires a set Elt of elements. We can then define

[3]In Maude, as in OBJ3, it is possible to declare several attributes of this kind for an operator, including also associativity and identity, and then do rewriting modulo such properties.

```
fmod REVERSE[X :: TRIV] is
  sort Tree .
  subsort Elt < Tree .
  op _^_ : Tree Tree -> Tree .
  op rev : Tree -> Tree .
  var E : Elt .
  vars T T' : Tree .
  eq rev(E) = E .
  eq rev(T ^ T') = rev(T') ^ rev(T) .
endfm
```

Such a parameterized module can then be instantiated by providing an interpretation—called a *view*—mapping the parameter sort `Elt` to a sort in the module chosen as the actual parameter. For example, if we interpret `Elt` as the sort `Nat` in the `NAT` module, then we can obtain an instantiation equivalent to the module `NAT-REVERSE` in our first example by writing

```
make NAT-REVERSE is REVERSE[Nat] endmk
```

Another example of a parameterized functional module is the following module for lists whose elements belong to a set of elements. The set of elements is a parameter that can be instantiated to any set; therefore, as in the previous example, the parameter theory is the trivial theory `TRIV`.

```
fmod LIST[X :: TRIV] is
  protecting NAT BOOL .
  sort List .
  subsort Elt < List .
  op __ : List List -> List [assoc id: nil] .
  op length : List -> Nat .
  op remove_from_ : List List -> List .
  op _in_ : Elt List -> Bool .
  vars E E' : Elt .
  vars L L' : List .
  eq length(nil) = 0 .
  eq length(E L) = (s 0) + length(L) .
  eq remove nil from L = L .
  eq remove L from nil = nil .
  eq remove E from (E' L) = if E == E' then
       remove E from L else E' remove E from L fi .
  eq remove E L' from L = remove L' from (remove E from L) .
  eq E in nil = false .
  eq E in (E' L) = if E == E' then true
       else E in L fi .
endfm
```

Note that the "empty syntax" operator `__` has been declared *associative* and has the constant `nil` as its *identity* element. The boolean operator `==` compares two terms and evaluates to `true` if they both evaluate to the same result and to `false` otherwise; this operator is built-in, but for modules with Church-Rosser and terminating equations it is always possible to define it equationally[4]. Rewriting with this module is performed *modulo* associativity and identity; this means that we can disregard parentheses and that a `List` variable can match `nil`. For example, if we instantiate this module to form lists of natural numbers by writing

```
    make NAT-LIST is LIST[Nat] endmk
```

then the second equation for `length` will match the expression `length(s s 0)` modulo associativity and identity by matching `E` to `s s 0` and `L` to `nil`.

2.1.2 Sort Constraints

The expressiveness of the order-sorted type structure can be further increased by the declaration of axioms called *sort constraints* [78] (declared by the keyword `sct`) stating that a given functional expression has a sort smaller than anticipated. We illustrate this notion—which also appears in OBJ3 with a different syntax [53]—by the (parameterized) definition of sets as a subsort of multisets using a sort constraint. The example reuses the parameterized `LIST` module, adapting it to the present context by renaming the `nil` list to the `null` multiset and the `List` sort to the `MSet` sort using a module renaming construct, and by turning list concatenation into multiset union thanks to the addition of a commutativity attribute. Such reuse illustrates a flexible form of *module inheritance* that is available in Maude through its submodule, parameterization, and module expression mechanisms and is further discussed in Section 4.3. However, as it was also done in the FOOPS language [49], such inheritance at the module level is sharply distinguished from *class inheritance*, which will be discussed in Section 4.2 and which is a special case of *subsort inheritance*.

```
  fmod MSET[X :: TRIV] is
    using LIST[X]*(sort List to MSet, op nil to null) .
    sort Set .
    subsorts Elt < Set < MSet .
    op __ : MSet MSet -> MSet [comm] .
    op null : -> Set .
    op set : MSet -> Set .
    op _U_ : Set Set -> Set .
    op |_| : Set -> Nat .
    var E : Elt .
    var MS : MSet .
    vars S S' : Set .
    sct (E S) : Set if not(E in S) .
    eq set(null) = null .
    eq set(E) = E .
    eq set(E MS) = if (E in MS) then set(MS) else E set(S) fi .
    eq S U S' = set(S S') .
    eq | S | = length(S) .
  endfm
```

[4]Cf. Theorems 54 and 71 in [77].

The keyword `using` asserts that, in this importation of the (renamed) `LIST` module, semantic modifications are being introduced so that the original equality relation between data elements will be altered[5]. The renaming expression

```
LIST[X]*(sort List to MSet, op nil to null)
```

changes the syntax of the `LIST[X]` module by renaming its sort `List` to `MSet` and its operator `nil` to `null`.

The original list concatenation operator is now understood as a multiset union operator; the original operator was already associative and had `nil` (now renamed to `null`) as an identity. Now we add a *commutativity* axiom, so that two multisets are equal if one is a permutation of the other; this is accomplished by adding a `[comm]` attribute declaration to the imported multiset union operator. Of course, this changes the equality relation between data elements. It also changes the rewriting relation in the sense that concurrent rewriting in this module is performed *modulo* the associativity, commutativity and identity axioms for the operator `__`. Therefore, we can disregard parentheses and the order of the arguments. We call rewriting modulo associativity, commutativity and identity *ACI-rewriting*.

The subsort `Set` of sets is defined as the subsort of all multisets with no repeated elements. This is accomplished by means of four declarations: the subsort declaration `Set < MSet`; an operator declaration stating that `null` is a set; a subsort declaration `Elt < Set` stating that singleton multisets are sets; and a sort constraint stating that the addition of an element to a set yields also a set provided that the element was not already inside the original set.

In general, a *sort constraint* [78] is a conditional assertion of the form

$$t(x_1, \ldots, x_n) : s \quad \textit{if} \quad u_1(x_1, \ldots, x_n) = v_1(x_1, \ldots, x_n) \wedge \ldots \wedge u_k(x_1, \ldots, x_n) = v_k(x_1, \ldots, x_n)$$

where s is a sort and $t(x_1, \ldots, x_n)$ and the u_j, v_j are terms whose variables have sorts, say, $x_i : s_i$, $i = 1, \ldots, n$. We say that an order-sorted algebra A *satisfies* such a sort constraint if, for any assignment of values $a_i \in A_{s_i}$ to the variables x_i, such that for each $j = 1, \ldots, k$ the elements $u_{j_A}(a_1, \ldots, a_n)$ and $v_{j_A}(a_1, \ldots, a_n)$—obtained by evaluating the terms u_j and v_j in the algebra A under the assignment—are equal, then the element $t_A(a_1, \ldots, a_n)$ obtained by evaluating the term t in the algebra A under the assignment belongs to the set A_s. For the sort constraint in the above example, $k = 1$, with u_1 the term `not(E in S)`, and v_1 (left implicit) the term `true`. If the sort constraint's condition is empty ($k = 0$) we speak of an *unconditional sort constraint* and adopt the notation

$$t(x_1, \ldots, x_n) : s$$

It is a fortunate fact that the class of order-sorted algebras satisfying a given set of equations and sort constraints has an initial algebra [76]. Of course, the *mathematical semantics* of (unparameterized) functional modules containing such sort constraint declarations is precisely such an initial algebra; a parameterized version of such an initiality result is what applies to the above multiset and set example. This means what we would naturally expect, i.e., that for each instantiation of the above module to a given set `X` of elements, the sort `MSet` consists of finite multisets of elements of `X`, the sort `Set` consists of finite sets of elements of `X`, and the data type operations between such sorts behave as desired.

[5]In general, further modifications by addition of new data elements to old sorts could also occur in a module imported with a `using` declaration; i.e., both addition of "junk" (new data elements) and of "confusion" between old data elements can take place in a `using` module importation.

In particular, the following additional operations have been defined: a function `set` that turns each multiset into a set by removing duplicated elements; a set union function `_U_`; and a set cardinality function `|_|`. Of course, a variety of other set-theoretic operations could have been defined similarly if desired.

2.2 System Modules

Maude system modules perform concurrent rewriting computations in exactly the same way as functional modules; however, their behavior is not functional. Consider the following module, `NAT-CHOICE`, which adds a nondeterministic choice operator to the natural numbers.

```
mod NAT-CHOICE is
  extending NAT .
  op _?_ : Nat Nat -> Nat .
  vars N M : Nat .
  rl N ? M => N .
  rl N ? M => M .
endm
```

The intuitive *operational behavior* of this module is quite clear. Natural number addition remains unchanged and is computed using the two rules in the `NAT` module. Notice that any occurrence of the choice operator `_?_` in an expression can be eliminated by choosing either of the arguments. In the end, we can reduce any ground expression to a natural number in Peano notation. The *mathematical semantics* of the module is much less clear. If we adopt any semantics in which the models are algebras satisfying the rules as equations—in particular an initial algebra semantics—it follows by the rules of equational deduction with the two equations in `NAT-CHOICE` that

```
N = M
```

i.e., everything collapses to one point. Therefore, the declaration `extending NAT`, whose meaning is that two distinct natural numbers in the submodule `NAT` are not identified by the new equations introduced in the supermodule `NAT-CHOICE`, i.e., that no "confusion" is introduced in the old data, is violated in the worse possible way by this semantics; yet, the operational behavior in fact respects such a declaration. To indicate that this is not the semantics intended, system modules are distinguished from functional modules by means of the keyword `mod`, instead of the previous `fmod`. Similarly, a new keyword `rl` is used for rewrite rules—instead of the usual `eq` before each equation—and the equal sign is replaced by the new sign "`=>`" to suggest that `rl` declarations must be understood as "rules" and not as equations in the usual sense. At the operational level the equations introduced by the keyword `eq` in a functional module are also implemented as rewrite rules; the difference however lies in the *mathematical semantics* given to the module, which for modules like the one above should *not* be the usual initial algebra semantics. We need a logic and a model theory that are the perfect match for this problem. For this solution to be in harmony with the old one, the new logic and the new model theory should *generalize* the old ones.

System modules can also be parameterized. For example, we could have defined a parameterized module with a nondeterministic choice operator

```
mod TICKET is
  sorts Place Marking .
  subsort Place < Marking .
  ops $,q,t1,t2 : -> Place .
  op __ : Marking Marking -> Marking
             [assoc comm id: null] .
  rl b-t1 : $ => t1 q q .
  rl b-t2 : $ => t2 q .
  rl change : $ => q q q q .
  rl b'-t1 : q q => t1 .
  rl b'-t2 : q q q => t2 .
endm
```

figure=/homes/dumas/winkler/text/fig/petrin.ps

Figure 2: A Petri net and its code in Maude.

```
mod CHOICE[X :: TRIV] is
  op _?_ : Elt Elt -> Elt .
  vars A B : Elt .
  rl A ? B => A .
  rl A ? B => B .
endm
```

and could have obtained a module equivalent to `NAT-CHOICE` by means of the module expression

```
make NAT-CHOICE is CHOICE[Nat] endmk
```

Another interesting example of a system module that illustrates both Maude's expressiveness and the generality of the concurrent rewriting model is the Petri net in Figure 2, which represents a machine to buy subway tickets. With a dollar we can buy a ticket $t1$ by pushing the button $b-t1$ and get two quarters back; if we push $b-t2$ instead, we get a longer distance ticket $t2$ and one quarter back. Similar buttons allow purchasing the tickets with quarters. Finally, with one dollar we can get four quarters by pushing *change*. The corresponding system module, `TICKET`, is given in the same figure. Note that the rules in this module are *labelled* by the name of the transition which they represent. A key point about this module is that the operator `__`—corresponding to *multiset union*—has been declared *associative*, *commutative*, and having an *identity* element `null`, and that rewriting is performed modulo those axioms, i.e., it is *ACI-rewriting*. In this example, *ACI*-rewriting captures exactly the concurrent computations of the Petri net. Suppose, for example, that we begin in a state with four quarters and two dollars. Then, by first concurrently pushing the buttons b'–$t1$ and b–$t2$, and then concurrently pushing the buttons b'–$t2$ and b–$t2$ we end up with a ticket for the shorter distance, three tickets for the longer distance and a quarter, as shown in the two steps of concurrent *ACI*-rewriting below:

`q q q q $ $` $\longrightarrow$ `q q t1 t2 q $` $\longrightarrow$ `t2 t1 t2 t2 q`.

As in the `NAT-CHOICE` example, this example also shows that initial algebra semantics is entirely inadequate to handle system modules with a nonfunctional behavior. In this case, interpreting the rules as equations would force the nonsensical identification of the three states above.

State	$\longleftrightarrow$	*Term*
Transition	$\longleftrightarrow$	*Rewriting*
Distributed Structure	$\longleftrightarrow$	*Algebraic Structure*

Figure 3: System-oriented interpretation of concurrent rewriting.

System modules denote *concurrent systems*, not algebras, and rewriting logic is a logic that expresses directly the concurrent computations of such systems.

Indeed, the passage from functional modules to system modules involves a fundamental change in perspective, so that basic notions that previously had a very familiar interpretation in functional terms have now to be reinterpreted in a very different way. In this new interpretation, *terms* are no longer understood as *functional expressions*, but as *structured states*, where the structure of the state is given by the operators that happen to appear in the term and by the structural axioms that they enjoy. For example, a Petri net marking is a state having a multiset structure given by a binary multiset union operator that enjoys ACI structural axioms. Similarly, an expression such as `(3 ? 5) ? (7 ? 12)` is a state having a binary tree structure. The algebraic structure of the state—as a multiset, binary tree, or whatever—is precisely what makes the state *distributed*, i.e., coincides with its *distributed structure*, and makes concurrency possible.

In the same way, *rewriting* is no longer seen as *functional evaluation* by equational deduction, but as *transition in a system*. This is clearly illustrated by the rewrite rules for the Petri net example, and also by the nondeterministic choice rules for the `NAT-CHOICE` example, where the final states are numbers. The states' algebraic—and therefore distributed—structure makes possible for many rewritings to occur *concurrently*, i.e., rewritings are *local* transitions of a distributed state that happen independently of each other. Figure 3 summarizes this discussion.

3 Rewriting Logic

Rewriting logic is defined, and concurrent rewriting is formalized as deduction in such a logic.

3.1 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 Σ of function symbols is a ranked alphabet $\Sigma = \{\Sigma_n \mid n \in \mathbb{N}\}$. A Σ-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_Σ the Σ-algebra of ground Σ-terms, and by $T_\Sigma(X)$ the Σ-algebra of Σ-terms with variables in a set X. Similarly, given a set E of Σ-equations, $T_{\Sigma,E}$ denotes the Σ-algebra of equivalence classes of ground Σ-terms modulo the equations E (i.e., modulo provable equality using the equations E); in the same way, $T_{\Sigma,E}(X)$ denotes the Σ-algebra of equivalence classes of Σ-terms with variables in X modulo the equations E. Let $t =_E t'$ denote the congruence modulo E of two terms t, t', and 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 $\overline{a}$, or, to emphasize the length of the sequence, by $\overline{a}^n$. With this notation, $t(u_1/x_1, \ldots, u_n/x_n)$ can be abbreviated to $t(\overline{u}/\overline{x})$.

3.2 The Rules of Rewriting Logic

We are now ready to introduce the new logic that we are seeking, which we call *rewriting logic*. A *signature* in this logic is a pair (Σ, E) with Σ a ranked alphabet of function symbols and E a set of Σ-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 (see, e.g., [56, 33]).

Given a signature (Σ, E), *sentences* of the logic are sequents of the form $[t]_E \longrightarrow [t']_E$ with t, t' Σ-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.

A (*labelled*) *rewrite theory*[6] $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-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*[7]. We understand a rule $(r, ([t], [t']))$ as a labelled 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[8] $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, or in abbreviated notation $r : [t(\overline{x}^n)] \longrightarrow [t'(\overline{x}^n)]$. □

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 $[t] \in T_{\Sigma,E}(X)$,

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

[6]We consciously depart from the standard terminology, that would call $\mathcal{R}$ a *rewrite system*. The reason for this departure is very specific. We want to keep the term "rewrite system" for the *models* of such a theory, which will be defined in Section 8 and which really are systems with a dynamic behavior. Strictly speaking, $\mathcal{R}$ is not a system; it is only a static, linguistic, *presentation* of a class of systems—including the initial and free systems that most directly embody our dynamic intuitions about rewriting.

[7]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 [72] 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 Maude examples presented in this paper.

[8]Note that, in general, the set $\{x_1, \ldots, x_n\}$ will depend on the representatives t and t' chosen; therefore, we allow any possible such qualification with explicit variables.

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

$$\frac{[t_1] \longrightarrow [t'_1] \quad \dots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \dots, t_n)] \longrightarrow [f(t'_1, \dots, t'_n)]}$$

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

$$\frac{[w_1] \longrightarrow [w'_1] \quad \dots \quad [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}$$

4. **Transitivity**.

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

Equational logic (modulo a set of axioms E) is obtained from rewriting logic by adding the following rule:

5. **Symmetry**.

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

With this new rule, sequents derivable in equational logic are *bidirectional*; therefore, in this case we can adopt the notation $[t] \leftrightarrow [t']$ throughout and call such bidirectional sequents *equations.*

3.3 Concurrent Rewriting as Deduction

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.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, a (Σ, E)-sequent $[t] \longrightarrow [t']$ is called:

- a 0-*step concurrent* $\mathcal{R}$-*rewrite* iff it can be derived from $\mathcal{R}$ by finite application of the rules 1 and 2 of rewriting deduction (in which case $[t]$ and $[t]'$ necessarily coincide);
- a *one-step concurrent* $\mathcal{R}$-*rewrite* iff it can be derived from $\mathcal{R}$ by finite application of the rules 1-3, with at least one application of rule 3; if rule 3 is applied exactly once, we then say that the sequent is a one-step *sequential* $\mathcal{R}$-rewrite;
- 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.

We call the rewrite theory $\mathcal{R}$ *sequential* if all one-step $\mathcal{R}$-rewrites are necessarily sequential. A sequential rewrite theory $\mathcal{R}$ is in addition called *deterministic* if for each $[t]$ there is at most one one-step (necessarily sequential) rewrite $[t] \longrightarrow [t']$. $\Box$

Example 1 All rewrite steps in Figure 1 are one-step concurrent rewrites, but none are sequential. For example, the first such step can be obtained by first applying replacement twice for the second rule in `NAT` to 0-step rewrites given by the substitutions $(\texttt{N} \mapsto \texttt{0}, \texttt{M} \mapsto \texttt{s0})$, and $(\texttt{N} \mapsto \texttt{0}, \texttt{M} \mapsto \texttt{0})$, to get sequents

$$[\texttt{s0+ss0}] \longrightarrow [\texttt{ss(0+s0)}], \qquad [\texttt{s0+s0}] \longrightarrow [\texttt{ss(0+0)}];$$

figure=/homes/dumas/winkler/text/fig/c.ps

Figure 4: The Church-Rosser property.

then, applying congruence to each of these sequents and appropriate 0-step rewrites to get sequents

$$[\texttt{(s0+ss0)^s0}] \longrightarrow [\texttt{ss(0+s0)^s0}], \qquad [\texttt{0^(s0+s0)}] \longrightarrow [\texttt{0^ss(0+0)}];$$

and finally, applying replacement to those two sequents for the second rule in the `REVERSE` module. Note that transitivity was never used. □

The usual notions of confluence, termination, normal form, etc., as well as the well known Church-Rosser property of confluent rules, remain unchanged when considered from the perspective of concurrent rewriting [72].

Specifically, we call a rewrite theory $\mathcal{R}$ *terminating* if there is no infinite chain of one-step rewrites (whether sequential or concurrent)

$$[t] \longrightarrow [t_1] \longrightarrow \ \ldots \ \longrightarrow [t_n] \longrightarrow \ \ldots$$

We say that $[t']$ is an $\mathcal{R}$-*normal form* of $[t]$ if $[t] \longrightarrow [t']$ is an $\mathcal{R}$-rewrite and there does not exist any one-step $\mathcal{R}$-rewrite of the form $[t'] \longrightarrow [t'']$. If each $[t]$ has at least one normal form, we call the theory $\mathcal{R}$ *weakly terminating.*

We say that a rewrite theory $\mathcal{R}$ is *Church-Rosser* or *confluent* if given any two concurrent rewrites $[t] \longrightarrow [t']$, $[t] \longrightarrow [t'']$, there is a $[t''']$ and concurrent rewrites $[t'] \longrightarrow [t''']$, $[t''] \longrightarrow [t''']$. This situation is shown in Figure 4. Likewise, we call $\mathcal{R}$ *ground Church-Rosser* when the property is only asserted for equivalence classes of ground terms. As already mentioned, the equations in Maude's functional modules are expected to be Church-Rosser, but system modules and object-oriented modules (see Section 4.1) are not expected to be Church-Rosser and typically they are not so. Indeed, the Church-Rosser property is a mark of functionality, since it guarantees that any term has at most one normal form, which can be interpreted as the result of its functional evaluation.

3.4 The Meaning of Rewriting Logic

A logic worth its salt should be understood as a method of correct reasoning about some class of entities, not as an empty formal game. For equational logic, the entities in question are sets, functions between them, and the relation of identity between elements. For rewriting logic, the entities in question are *concurrent systems* having *states*, and evolving by means of *transitions*. The *signature* of a rewrite theory describes a particular structure for the states of a system—e.g., multiset, binary tree, etc.—so that its states can be distributed according to such a structure. The *rewrite rules* in the theory describe which *elementary local transitions* are possible in the distributed state by concurrent local transformations. What the rules of rewriting logic allow us to reason correctly about is which *general* concurrent transitions are possible in a system satisfying such a description. Clearly, concurrent systems should be the *models* giving a semantic interpretation to rewriting logic, in the same way that algebras are the models giving a semantic interpretation to equational logic. A precise account of the model theory of rewriting logic, giving rise to an initial model semantics for Maude modules and fully consistent with the above system-oriented interpretation, is given in Section 8.

$$\begin{array}{lclcl} \textit{State} & \leftrightarrow & \textit{Term} & \leftrightarrow & \textit{Proposition} \\ \textit{Transition} & \leftrightarrow & \textit{Rewriting} & \leftrightarrow & \textit{Deduction} \\ \textit{Dist. Struct.} & \leftrightarrow & \textit{Alg. Struct.} & \leftrightarrow & \textit{Prop. Conn.} \end{array}$$

Figure 5: The meaning of rewriting logic.

Therefore, in rewriting logic a sequent $[t] \longrightarrow [t']$ should not be read as "$[t]$ *equals* $[t']$," but as "$[t]$ *becomes* $[t']$." Clearly, rewriting logic is a logic of *becoming* or *change*, not a logic of equality in a static Platonic sense. The apparently innocent step of adding the symmetry rule is in fact a *very strong* restriction, namely assuming that *all change is reversible*, thus bringing us into a timeless Platonic realm in which "before" and "after" have been identified.

A related observation is that $[t]$ should not be understood as a *term* in the usual first-order logic sense, but as a *proposition*—built up using the *propositional connectives* in Σ—that asserts being in a certain *state* having a certain *structure*. However, unlike most other logics, the logical connectives Σ and their structural properties E are entirely *user-definable*. This provides great flexibility for considering many different state structures and makes rewriting logic very general in its capacity to deal with many different types of concurrent systems. This generality is further discussed in Section 4.8, and is treated in greater length in [72].

In summary, the rules of rewriting logic are rules to reason about *change in a concurrent system*[9]. They allow us to draw valid conclusions about the evolution of the system from certain basic types of change known to be possible thanks to the rules R. Our present discussion is summarized in Figure 5, which extends Figure 3 by adding each concept's logical counterpart.

4 A Logical Theory of Concurrent Objects

We are now ready to present a logical theory of concurrent objects based on rewriting logic deduction modulo ACI. The key idea is to conceptualize the distributed state of a concurrent object-oriented system—called a *configuration*—as a multiset of objects and messages that evolves by concurrent ACI-rewriting using rules that describe the effects of *communication events* between some objects and messages. Therefore, we can view concurrent object-oriented computation as *deduction* in rewriting logic; in this way, the configurations S that are *reachable* from a given initial configuration S_0 are exactly those such that the sequent $S_0 \longrightarrow S$ is *provable* in rewriting logic using the rewrite rules that specify the behavior of the given object-oriented system.

An *object* in a given state is 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 basic syntax and sort structure for attributes, objects, messages and configurations are given by the modules `ATTRIBUTES`

[9]Since rewriting logic is a logic of change, it has some similarities with Girard's linear logic [39, 40] (see also [65] for a survey of work on the relationship between linear logic and concurrency). In fact, from the perspective of rewriting logic the quantifier-free fragment of linear logic appears as a particular choice of user-definable connectives that are expressed in a very direct and natural way within rewriting logic in a conservative way [63].

and CONFIGURATION below; the first is a functional module defining the data type of attributes, and the second is a system module comprising all the entities that make up configurations. Before introducing such modules, we introduce an auxiliary parameterized module MAP that iterates a given parameter function over a multiset. This auxiliary module is used later to extract the set of attribute identifiers occurring in a set of attributes.

```
fth FUNCTION is
  sorts A B .
  op f: A -> B .
endft

fmod MAP[F :: FUNCTION] is
  protecting MSET[A]*(sort MSet to AMSet, sort Set to ASet) .
  protecting MSET[B]*(sort MSet to BMSet, sort Set to BSet) .
  op map : AMSet -> BMSet .
  var X : A .
  var AMS : AMSet .
  eq map(null) = null .
  eq map(X AMS) = f(X) map(AMS) .
endfm
```

We assume an already existing functional module ID of identifiers containing the sorts OId, CId and AId of object, class and attribute identifiers respectively (all of which are particular subsorts of the very general sort Value).

```
fmod ATTRIBUTES is
  protecting ID . *** provides OId, CId and AId
  sorts Attribute Attributes Value .
  subsorts OId CId AId < Value .
  op (_:_) : AId Value -> Attribute .
  op aid : Attribute -> AId .
  var A : AId .
  var V : Value .
  eq aid(A : V) = A .

  protecting MAP[aid]*(sort AMSet to AttMSet, sort ASet to AttSet,
       sort BMSet to AIdMSet, sort BSet to AIdSet,
       op (__) : AMSet AMSet -> AMSet to (_,_)) .

  subsorts Attribute < Attributes < AttSet .
  op null : -> Attributes .
  var ATT : Attribute .
  var ATTS : Attributes .
  sct (ATT, ATTS) : Attributes if not aid(ATT) in map(ATTS) .
endfm
```

The key sort being defined in the above module is the sort Attributes that will be used for the attributes of an object. A data element of that sort is a set of attributes[10]—where we have now adopted a notation that separates the different elements in the set by commas, as described in the renaming for the instantiation `MAP[aid]` of the `MAP` module that maps the parameter operator `f` to `aid`—but it must in addition satisfy the condition that *no attribute identifier can ever appear twice*; this would for example be violated by the set of attributes

```
a: 3, b: true, a: 7
```

This condition is guaranteed by means of three assertions: the subsort declaration `Attribute < Attributes`; the operator declaration `null : -> Attributes`; and the sort constraint.

We are now ready for introducing objects, messages and configurations in the `CONFIGURATION` module below. Some attributes of an object can be *hidden*. Messages belong to a sort `Msg`; some very general messages that can be used to query objects and to get responses are also defined below.

```
mod CONFIGURATION is
  protecting ATTRIBUTES .
  sorts Configuration Object Msg .
  subsorts Object Msg < Configuration .
  op __ : Configuration Configuration -> Configuration
                               [assoc comm id: null] .
  op <_:_|_> : OId CId Attributes -> Object .
  op _._replyto_ : OId AId OId -> Msg .
  op to_,_._is_ : OId OId AId Value -> Msg .
  var C : CId .
  var ATTS : Attributes .
  var AIDST : AIdSet .
  vars A B : OId .
  var AID : AId .
  var V : Value .
  rl (A . AID replyto B) < A : C | AID : V, ATTS > =>
       < A : C | AID : V, ATTS > (to B,  A . AID is V)
       if not(hidden in map(AID : V, ATTS)) .
  rl (A . AID replyto B) < A : C | AID : V, hidden : AIDST, ATTS > =>
       < A : C | AID : V, hidden : AIDST, ATTS >
       (to B, A . AID is V) if not(AID in AIDST) .
endm
```

The *ACI*-operator `__` plays a role entirely similar to that played by the operator with the same syntax used for Petri nets, namely that of structuring the distributed state as a multiset. Objects with hidden attributes have an attribute of the form `(hidden : AIDST)` with `AIDST` a set of attribute identifiers specifying what attributes are hidden. Two very general kinds of messages are introduced. One that permits requesting that the value of an attribute identifier of a given object is sent to another object, and another for honoring such a request. The two rewrite rules specify the behavior of an object upon receiving a request for the value of one of

[10]The convenience of making the "union of attributes" operator `_,_` into an *ACI*-operator will become apparent when we discuss inheritance issues in Section 4.2 and is also illustrated by the rules for `CONFIGURATION` below.

its attributes. If the attribute in question does not appear in the object, or is among those hidden in the object, or is itself the attribute identifier `hidden`, then nothing happens, i.e., no rewrite rule applies[11]; otherwise, the attribute's value is sent to the requesting object.

The type structure provided by the above signature is still rather unconstrained. For example, the definition of a class C of objects introduced in a given object-oriented module (see Section 4.1) will have the effect of constraining the attribute identifiers of objects in that class to contain a specific set $\{a_1, \ldots, a_n\}$ of attribute identifiers, and a subclass definition enlarges such a set. Similarly, the sort `Value` is typically the supersort of a possibly quite complex collection of (functional) algebraic data types, whose computations can also be specified by rewrite rules introduced in appropriate functional submodules of the system, but could in some cases contain also nonfunctional entities such as objects or entire configurations (see Section 4.6). In a class definition, the values v over which an attribute a ranges are typically forced to be in a given subsort of `Value`. Such tightening of the type structure to exactly reflect the type requirements of a given object-oriented system is discussed in Section 4.2.

4.1 Object-Oriented Modules

In Maude, concurrent object-oriented systems can be defined by means of *object-oriented modules*—introduced by the keyword `omod`—using a syntax more convenient than that of system modules because it assumes acquaintance with basic entities such as objects, messages and configurations, and supports linguistic distinctions appropriate for the object-oriented case; however, the syntax and semantics of object-oriented modules can be reduced to that of system modules as explained in Section 4.2.

For example, the `ACCNT` object-oriented module below specifies the concurrent behavior of objects in a very simple class `Accnt` of bank accounts, each having a `bal`(ance) attribute, which may receive messages for crediting or debiting the account, or for transferring funds between two accounts. We assume an already given functional module `REAL` for real numbers with a subsort relation `NNReal < Real` corresponding to the inclusion of the nonnegative reals (i.e., reals greater or equal than zero) into the reals, and with an ordering predicate `_>=_`.

```
omod ACCNT is
  protecting REAL .
  class Accnt | bal: NNReal .
  msgs credit debit : OId NNReal -> Msg .
  msg transfer_from_to_ : NNReal OId OId -> Msg .
  vars A B : OId .
  vars M N N' : NNReal .
  rl credit(A,M) < A : Accnt | bal: N > => < A : Accnt | bal: N + M > .
  rl debit(A,M) < A : Accnt | bal: N > => < A : Accnt | bal: N - M >
       if N >= M .
  rl transfer M from A to B
       < A : Accnt | bal: N > < B : Accnt | bal: N' > =>
       < A : Accnt | bal: N - M > < B : Accnt | bal: N' + M >
       if N >= M .
endom
```

[11]If desired, one could of course specify additional rules to send back an appropriate error message instead.

Figure 6: Concurrent rewriting of bank accounts.

After the keyword `class`, the name of the class—in this case `Accnt`—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 sort inside which the values of such an attribute identifier must range in the given class. In this example, the only attribute of an account is its `bal`(ance), which is declared to be a value in `NNReal`. The three kinds of messages involving accounts are `credit`, `debit`, and `transfer` messages, whose user definable syntax is introduced by the keyword `msg`. The rewrite rules specify in a declarative way the behavior associated with the `credit`, `debit`, and `transfer` messages.

The multiset structure of the configuration provides the top level distributed structure of the system and allows concurrent application of the rules. For example, Figure 6 provides a snapshot in the evolution by concurrent rewriting of a simple configuration of bank accounts. To simplify the picture, the arithmetic operations required to update balances have already been performed. However, the reader should bear in mind that the values in the attributes of an object can also be computed by means of rewrite rules, and this adds yet another important level of concurrency to a concurrent object-oriented system, which might be called *intra-object concurrency*[12]. Intra-object concurrency seems to be absent from the standard models and languages for concurrent object-oriented programming, where only *inter-object concurrency* is considered.

The system evolves by concurrent rewriting (modulo *ACI*) of the configuration using the rewrite rules of the system, whose lefthand and righthand sides may—as illustrated in the example above—involve patterns for several objects and messages. Intuitively, we can think of messages as "travelling" to come into contact with the objects to which they are sent and then causing "communication events" by application of rewrite rules. In the model, this travelling is accounted for in a very abstract way by the *ACI* axioms. This abstract level supports both synchronous and asynchronous communication and provides great freedom and flexibility to consider a variety of alternative implementations at lower levels. Such abstraction from implementation details makes possible high level reasoning about concurrent object-oriented programs and their semantics without having to go down into the specific details of how communication is actually implemented.

Another example of an object-oriented module is a module for FIFO buffers of bounded size. The set of elements to be stored in the buffer is a parameter; the other parameter is the size of the buffer which is specified by the (functional) parameter theory

```
fth NAT* is
  protecting NAT .
  op k : -> NzNat .
endft
```

whose models are choices of a nonzero natural number `k`. The bounded buffer module is as follows

```
omod BD-BUFF[X :: TRIV, K :: NAT*] is
  protecting LIST[X] .
```

[12]The eight queens example in Section 6.1 provides a good illustration of intra-object concurrency.

```
    class BdBuff | contents: List [hidden] .
    msg put_in_ : Elt OId -> Msg .
    msg getfrom_replyto_ : OId OId -> Msg .
    msg to_elt-in_is_ : OId OId Elt -> Msg .
    vars B I : OId .
    var E : Elt .
    var Q : List .
    rl (put E in B) < B : BdBuff | contents: Q > =>
         < B : BdBuff | contents: E Q >
         if length(Q) < k .
    rl (getfrom B replyto I)
         < B : BdBuff | contents: Q E > =>
         < B : BdBuff | contents: Q >
         (to I elt-in B is E) .
  endom
```

The only attribute of a buffer is its `contents`, which is a list of elements. Since the contents should not be visible outside the buffer, this attribute has been declared `[hidden]`; this means that no other object can send a message requesting the entire contents of the buffer because messages of that kind are ruled out for hidden attributes. The two types of communication events that are possible are specified by the two rules of the module. If an arbitrary object `I` possesses the name `B` of a buffer, then it is possible for that object to either send a message `(put E in B)` to put the element `E` in `B`, or to send a message `(getfrom B replyto I)` to `B`, and the last rule specifies that, when `B` has a nonempty queue, it will send the first element of its queue to `I` by means of the message `(to I elt-in B is E)`.

The two rules in the bounded buffer module provide a simple declarative solution to the problem of specifying the appropriate behavior of a bounded buffer that receives a `put` message when it is full or a `get` message when it is empty. The implicit effect of the rules is that the corresponding messages "float" in the configuration until the appropriate conditions for the buffer hold; if additional error handling is desired, this can be specified by adding more rules. By contrast, a language like ABCL/1 [104] requires introducing a special "waiting mode" for objects and a corresponding "select construct" to reactivate the object appropriately after such waiting. The simplicity of the treatment afforded by rewrite rules in this example exemplifies a general fact, namely that using rewrite rules there is no need for any explicit "synchronization code." This permits avoiding the "inheritance anomaly" [74], a subject further discussed in Section 4.2.

Specific instances of the bounded buffer module can then be obtained by providing appropriate *views* for its two parameter theories. For example, a buffer of length 4,096 holding characters of sort `Char` that we assume already introduced in a module `CHAR` can be defined by

```
  make BD-CHAR-BUFF is BD-BUFF[Char,view to NAT is op k to 4096 . endv] endmk
```

where the first abbreviated view maps `Elt` to `Char`, and the second view sets the value of the bound `k` to be `4,096` (we assume standard notation for numbers here).

4.1.1 General Form of the Rules

In Maude, the general form required of rewrite rules used to specify the behavior of an object-oriented system is as follows:

$$
\begin{aligned}
(\dagger)\quad & M_1 \ldots M_n \ \langle O_1 : C_1 \,|\, \mathit{atts}_1 \rangle \ldots \langle O_m : C_m \mid \mathit{atts}_m \rangle \\
& \longrightarrow \langle O_{i_1} : C'_{i_1} \mid \mathit{atts}'_{i_1} \rangle \ldots \langle O_{i_k} : C'_{i_k} \mid \mathit{atts}'_{i_k} \rangle \\
& \quad \langle Q_1 : D_1 \mid \mathit{atts}''_1 \rangle \ldots \langle Q_p : D_p \mid \mathit{atts}''_p \rangle \\
& \quad M'_1 \ldots M'_q \\
& \quad \mathit{if}\ C
\end{aligned}
$$

where $k, p, q \geq 0$, the Ms are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is the rule's condition. A rule of this kind expresses a *communication event* in which n messages and m distinct objects participate. The *outcome* of such an event is as follows:

- the messages $M_1, \ldots, M_n$ disappear;
- the *state* and possibly even the *class* of the objects $O_{i_1}, \ldots, O_{i_k}$ may change;
- all other objects O_j vanish;
- new objects $Q_1, \ldots, Q_p$ are created;
- new messages $M'_1, \ldots, M'_q$ are sent.

Notice that, since some of the attributes of an object—as well as the parameters of messages—can contain object names, very complex and dynamically changing patterns of communication can be achieved by rules of this kind. Notice also that when $k = p = q = 0$ all the objects vanish, and no new objects or messages are created.

In addition, all rules must satisfy the property that *rewriting of a configuration without repeated object names leads to a configuration without repeated object names*. In other words, we are only interested in configurations in which there is a *set* of objects, not a multiset, and we never want to reach a configuration in which two objects have the same name; however, there is no problem in allowing configurations in which identical copies of a message have been sent, perhaps as the outcome of different communication events. A *necessary condition* required for this property to hold for a rule is that if in a *ground instance* of the rewrite rule the instances of the object names $O_1, \ldots, O_m$ are all different, then the instances of the object names $Q_1, \ldots, Q_m$ are also all different and different from the Os. Sufficient conditions to guarantee the uniqueness of objects are discussed in Section 4.4.

Although the above discussion suffices for the moment, more has to be said on the rules of object-oriented systems. Section 4.2 explains how rules are inherited and gives notational conventions that simplify the writing of rules. Also, since the above form of rules is very general, it is important to identify useful commonly occurring subcases that allow much more efficient implementation; this is the theme of Section 5 in which the Simple Maude sublanguage is introduced.

4.1.2 Synchrony, Asynchrony, and Autonomous Objects

Given the general form (†) of rewrite rules representing communication events in an object-oriented system, it is possible for one, none, or several objects to appear as participants in the lefthand sides of rules. If only one object appears in the lefthand side, we call such a communication event *asynchronous*, whereas if several objects are involved we call it *synchronous* and say that the objects in question are forced to *synchronize* in the event. For example, the rules for crediting and debiting accounts describe asynchronous communication events, whereas the rule for transferring funds between two accounts forces them to synchronize.

Note that a particular case allowed by the general form (†) of the rewrite rules is that in which no messages at all appear in the lefthand side. This gives rise to a rather striking mode of activity, namely that of objects that—on their own or synchronizing with other objects—change their state and/or send messages to other objects *without any external prompting by messages*. We call objects that can evolve without receiving messages by means of rules of this type *autonomous objects*.

Note that objects, whether autonomous or not, can in some cases exhibit a never-ending pattern of activity. For example, a *clock* object having a *time* attribute (say, a natural number) may update its time by sending a `tick` message to itself and having a "ticking rule" that increases the time by one unit and sends another `tick` message each time a `tick` message is received.

```
tick(C) < C : Clock | time: T > => < C : Clock | time: T + 1 > tick(C) .
```

Of course, we could define clocks in a different way as autonomous objects by eliminating the `tick` message and giving instead a rule

```
< C : Clock | time: T > => < C : Clock | time: T + 1 > .
```

The general form (†) of rules is too general for efficient implementation purposes and we may very well wish to seek additional restrictions under which an efficient implementation can be attained at the expense of some loss in expressiveness. This topic, and a particular set of restrictions leading to a sublanguage called Simple Maude having only asynchronous communication that we have studied jointly with Timothy Winkler in [83], are discussed in Section 5. The general idea is to consider Maude as a wide-spectrum language such that more expressive specification and rapid prototyping can be carried out in Maude, but where efficient execution assumes the restrictions in Simple Maude. One can then adopt a transformational approach to move from specifications to—possibly inefficient—prototypes, and from prototypes to efficient code. We are currently investigating techniques of this kind in joint work with Patrick Lincoln and Timothy Winkler.

4.2 Class Inheritance and Reduction to System Modules

Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration `C < C'` in an object-oriented module `omod` $\mathcal{O}$ `endom` is just a particular case of a subsort declaration `C < C'`[13]. As we shall see, the effect of a subclass declaration is that the attributes, messages and rules of all the superclasses as well as the newly defined attributes,

[13]The reason why classes, although conceptually distinguished from other sorts, have in essence the same treatment for their subsort relations as any other sorts will become clearer after we discuss the reduction of object-oriented modules to system modules later in this section.

messages and rules of the subclass characterize the structure and behavior of the objects in the subclass.

For example, we can define an object-oriented module `CHK-ACCNT` of checking accounts introducing a subclass `ChkAccnt` of `Accnt` with a new attribute `chk-hist` recording the history of checks cashed in the account.

```
omod CHK-ACCNT is
  extending ACCNT .
  dfn ChkHist is LIST[2TUPLE[Nat,NNReal]] .
  class ChkAccnt | chk-hist: ChkHist .
  subclass ChkAccnt < Accnt .
  msg chk_#_amt_ : OId Nat NNReal -> Msg .
  var A : OId .
  vars M N : NNReal .
  var K : Nat .
  var H : ChkHist .
  rl (chk A # K amt M) < A : ChkAccnt | bal: N, chk-hist: H >
        => < A : ChkAccnt | bal: N - M, chk-hist: H << K ; M >> >
       if N >= M .
endom
```

Adopting the same convention as in OBJ3, the statement

```
  dfn ChkHist is LIST[2TUPLE[Nat,NNReal]] .
```

is an abbreviation for the statement

```
  protecting LIST[2TUPLE[Nat,NNReal]]*(sort List to ChkHist) .
```

which imports a data type of lists of 2-tuples (pairs denoted `<<_;_>>`) consisting of a natural number and a nonnegative real, and renames the principal sort `List` to `ChkHist`. The checking history of the account is then represented as a list of such pairs with the first number in the pair corresponding to the check number, and the second number corresponding to the check's amount.

The best way to understand classes and class inheritance in Maude is by making explicit the full structure of an object-oriented module which is left somewhat implicit in the syntactic conventions adopted for them. Indeed, although Maude's object-oriented modules provide a convenient syntax for programming object-oriented systems, their semantics can be reduced to that of system modules; in a sense we can regard the special syntax reserved for object-oriented modules as syntactic sugar. In fact, each object-oriented module `omod` $\mathcal{O}$ `endom` can be translated into a corresponding system module `mod` $\mathcal{O}\#$ `endm` whose semantics *is* by definition that of the original object-oriented module[14].

However, although Maude's object-oriented modules can in this way be reduced to system modules, there are of course important conceptual advantages provided by the syntax of object-oriented modules, because it allows the user to think and express his or her thoughts in object-oriented terms whenever such a viewpoint seems best suited for the problem at hand. Those conceptual advantages would be lost if only system modules were provided.

[14]For the moment, consider the semantics of the module in terms of concurrent rewriting. Section 8 gives a model-theoretic semantics for Maude modules that makes completely precise their intended semantics.

In the translation process, the most basic structure shared by all object-oriented modules is made explicit by the CONFIGURATION system module defined at the beginning of this section. The translation of a given object-oriented module extends this structure with the classes, messages and rules introduced by the module. For example, the following system module ACCNT# is the translation of the ACCNT module introduced earlier. Note that a subsort <Accnt of CId is introduced. The purpose of this subsort is to range over the class identifiers of the subclasses of Accnt. For the moment, no such subclasses have been introduced; therefore, at present the only constant of sort <Accnt is the class identifier[15] Accnt.

```
mod ACCNT# is
  extending CONFIGURATION .
  protecting REAL .
  sorts <Accnt Accnt .
  subsort Accnt < Object .
  subsort <Accnt < CId .
  subsort NNReal < Value .
  op Accnt : -> <Accnt .
  ops credit,debit : OId NNReal -> Msg .
  op transfer_from_to_ : NNReal OId OId -> Msg .
  vars A B : OId .
  vars M N N' : NNReal .
  var X : <Accnt .
  vars ATTS ATTS' : Attributes .
  sct < A : X | bal: N, ATTS > : Accnt .
  rl credit(A,M) < A : X | bal: N, ATTS > => < A : X | bal: N + M, ATTS > .
  rl debit(A,M) < A : X | bal: N, ATTS > => < A : X | bal: N - M, ATTS >
       if N >= M .
  rl transfer M from A to B
       < A : X | bal: N, ATTS > < B : X | bal: N', ATTS' > =>
       < A : X | bal: N - M, ATTS > < B : X | bal: N' + M, ATTS' >
       if N >= M .
endm
```

Objects of sort Accnt are defined by a sort constraint requiring that its class identifier has sort <Accnt and that one of its attributes is of the form (bal : N), with N of sort NNReal. Note that the rewrite rules originally introduced in the ACCNT module have been modified to make them applicable not only to objects whose class identifier is exactly Accnt, but also to other objects with class identifiers for subclasses of Accnt, which may in addition have other attributes, i.e., indeed to all the objects of the class Accnt. In other words, whenever a class identifier C appears in the lefthand side of a rule declared in an object-oriented module, we implicitly understand that a variable ranging over <C is meant instead. This way of inheriting rules was pointed out in Section 4.4 of [69]; I am indebted to Timothy Winkler for later

[15]Notice the slight ambiguity introduced by this notation, since now Accnt 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.

suggesting to me the elegant sort structure of the sorts `<C` as a better alternative to a more cumbersome identifier data type definition.

Note that the convention just described also involves leaving implicit a variable `ATTS`, ranging over the additional attributes that may appear in a subclass. In fact, we can further simplify the notation used in object-oriented modules by adopting, in addition, the convention of not mentioning in a given rule those attributes of an object that are not relevant for that rule. To explain this additional convention, let $\overline{a : v}$ denote the attribute-value pairs $a_1 : v_1, \ldots, a_n : v_n$, where the $\overline{a}$ are the attribute identifiers of a given class C having $\overline{s}$ as the corresponding sorts of values prescribed for those attributes. In this context, the v_i can be either terms (with or without variables) or variables of sort s_i.

The general convention is that in object-oriented modules we allow rules where the attributes appearing in the lefthand and righthand side patterns for an object O mentioned in the rule need not exhaust all the object's attributes, but can instead be in any two arbitrary subsets of the object's attributes[16]. We can picture this as follows

$$\ldots \langle O : C \mid \overline{al : vl}, \overline{ab : vb} \rangle \ldots \longrightarrow \ldots \langle O : C \mid \overline{ab : vb'}, \overline{ar : vr} \rangle \ldots$$

where $\overline{al}$ are the attributes appearing only on the *left*, $\overline{ab}$ are the attributes appearing on *both* sides, and $\overline{ar}$ are the attributes appearing only on the *right*. What this abbreviates in the corresponding reduction to a system module notation is a rule of the form

$$\ldots \langle O : X \mid \overline{al : vl}, \overline{ab : vb}, \overline{ar : x}, atts \rangle \ldots \longrightarrow \ldots \langle O : X \mid \overline{al : vl}, \overline{ab : vb'}, \overline{ar : vr}, atts \rangle \ldots$$

where X is a variable of sort $<C$, the $\overline{x}$ are new "don't care" variables and *atts* matches the remaining attribute-value pairs. The attributes mentioned only on the left are preserved unchanged, the original values of attributes mentioned only on the right don't matter, and all attributes not explicitly mentioned are left unchanged[17]. We can illustrate this convention with a simple example, namely a rule for a new type of message requesting the highest check number already cashed in a checking account

```
(highest-chk# A reply to B) < A : ChkAccnt | chk-hist: H >
   => < A : ChkAccnt | chk-hist: H >
       (to B highest-chk# A is max.1st(H))
```

where `max.1st` is an appropriately defined function that computes the highest number among those in the first components of pairs in the list. Note that the `bal` attribute is not mentioned at all, although of course it must be present in all objects in a subclass of `Accnt`. The rewrite rule for which the above rule is just a shorthand notation is

```
(highest-chk# A reply to B) < A : X | bal: M, chk-hist: H, ATTS >
   => < A : X | bal: M, chk-hist: H, ATTS >
       (to B highest-chk# A is max.1st(H))
```

[16]We assume that, as it is usually but not exclusively the case, the class of the object O does not change due to the rewrite; however, it should be possible to extend the present convention to some cases of interest in which the class does change.

[17]This notational convention generalizes a similar convention in [69] and has been developed in joint work with Timothy Winkler [83].

where X is a variable of sort <ChkAccnt, M is a variable of sort NNReal, H is a variable of sort ChkHist, and ATTS is a variable of sort Attributes.

Note that, since the CONFIGURATION module has been imported, besides the messages for crediting and debiting accounts, and for transferring funds between accounts, an account A with balance N can also receive messages of the form (bal . A replyto O) and will then respond to O by sending the message (to O bal . A is N). Since this capability is built in for arbitrary objects—unless the requested attribute has been declared hidden—it is never mentioned in the definition of an object-oriented module.

We are now ready to consider the full structure of subclasses. We can illustrate that structure by means of the reduction of the CHK-ACCNT module to its system module form.

```
mod CHK-ACCNT# is
  extending ACCNT# .
  sorts <ChkAccnt ChkAccnt  .
  subsort ChkAccnt < Accnt .
  subsort <ChkAccnt < <Accnt .
  dfn ChkHist is LIST[2TUPLE[Nat,NNReal]] .
  subsort ChkHist < Value .
  op ChkAccnt : -> <ChkAccnt .
  op chk_#_amt_ : OId Nat NNReal -> Msg .
  var A : OId .
  var X : <ChkAccnt .
  var ATTS : Attributes .
  vars M N : NNReal .
  var K : Nat .
  var H : ChkHist .
  sct < A : X | bal: N, chk-hist: H, ATTS > : ChkAccnt .
  rl (chk A # K amt M) < A : X | bal: N, chk-hist: H, ATTS >
      => < A : X | bal: N - M, chk-hist: H << K ; M >>, ATTS >
       if N >= M .
endm
```

Note that—in addition to a subsort declaration ChkAccnt < Accnt stating that checking accounts are accounts, i.e., the subclass relation—a subsort <ChkAccnt < <Accnt has been declared, having a class identifier ChkAccnt (introduced in a later operator declaration) in that subsort; i.e., the sort hierarchy for class identifiers mimics the class hierarchy, and has a class identifier constant (with the same name as that of the class) for each class in the corresponding point of the hierarchy.

An object in the class ChkAccnt must satisfy the sort constraint that its class identifier has sort <ChkAccnt and that it has at least two attributes, one called bal with a NNReal value, and another called chk-hist with a ChkHist value. The rewrite rule given in the original CHK-ACCNT module is interpreted here—according to the conventions already explained—in a form that can be inherited by subclasses of ChkAccnt that could be defined later. ChkAccnt itself inherits the rewrite rules for crediting and debiting accounts, and for transferring funds between accounts that had been defined for Accnt, and also the rules for requesting the value of its attributes which had been defined in the CONFIGURATION module.

In this example, there is only one class immediately above `ChkAccnt`, namely, `Accnt`. In general, however, a class C may be defined as a subclass of several classes $D_1, \ldots, D_k$, i.e., *multiple inheritance* is supported. Each class C has associated with it a family of pairs $\overline{a : s}$ each consisting of a different attribute identifier and a sort for its values. In case hidden attributes have been declared, we also associate with the class C the subset $a_{i_1}, \ldots, a_{i_j}$ of those attribute identifiers that are *hidden*. The objects of a class C not having any hidden attributes are then defined by a sort constraint of the form

$$\langle O : X \mid \overline{a : Y},\ \mathit{ATTS} \rangle : C$$

where the variable O has sort `OId`, X has sort $<C$, the variables $\overline{Y}$ have sorts $\overline{s}$, and the variable ATTS has sort `Attributes`. If hidden attributes $a_{i_1}, \ldots, a_{i_j}$ have been declared, the above sort constraint must be made *conditional* on having all the hidden attribute identifiers $a_{i_1}, \ldots, a_{i_j}$ inside the set denoted by the variable Y_q, where $a_q = \texttt{hidden}$. For example, the sort constraint for the class `BdBuff` of bounded buffers in the parameterized module `BD-BUFF`, which has its `contents` attribute hidden is

```
sct < O : X | contents: Q, hidden: S, ATTS > : BdBuff if (contents in S) .
```

with `O` of sort `OId`, `X` of sort `<BdBuff`, `Q` of sort `List`, and `ATTS` of sort `Attributes`.

If an attribute and its sort have already been declared in a superclass, they should not be declared again in the subclass. Indeed, all such attributes—whether hidden or not—are *inherited*, and if they were hidden in a superclass they remain hidden in the subclass. In the case of multiple inheritance, the only requirement that is made is that if an attribute a occurs in two different superclasses, then the sort attributed to it in each of those superclasses is the same[18]. In summary, a class inherits all the attributes, messages, and rules from all its superclasses. 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.

One caveat: in the case of *multiple* inheritance, it is in principle possible for a specific kind of message to have been introduced in two different superclasses, with quite different rules for handling such messages in each of them. As already mentioned, if a new class is defined as a subclass of each of those two superclasses, it will inherit the rules from both of them. In some cases this may be unproblematic, in the sense that everything behaves as expected because either the rules for one superclass specify the same behavior for that message as the rules of the other, or each set of rules specifies behavior for mutually exclusive circumstances, or, more generally, the two sets of rules agree on some cases and do not overlap on the remaining cases. The problem may arise when genuinely different behavior can result in the same situation depending on whether the rule applied belongs to one superclass or another, since this difference in behavior may be unintended. What this could probably indicate is a wrong use of the class inheritance mechanism on the part of the user. The right solution in such a situation may be to use *module inheritance* mechanisms instead (see Section 4.3) to obtain the desired behavior. In any case, the system should always warn the user whenever a potential source of unintended behavior arises due to the inheritance of messages from two unrelated superclasses.

[18]This condition could be relaxed to just requiring that the sorts in question have some common subsorts, and we could similarly allow that in a subclass the sort of an inherited attribute is restricted to a subsort. However, although somewhat more expressive, these relaxations would introduce some complications in the inheritance of rewrite rules. To simplify the exposition we do not treat here this more general case and restrict ourselves to the simpler sort assumptions just explained.

Rule inheritance in Maude solves the so-called *inheritance anomaly* [67, 90, 38] blocking the combination of inheritance and concurrency in object-oriented languages. The anomaly has to do with the serious difficulties often found for reusing in a subclass the code that handles the messages received by an object of a given class and performs appropriate actions. Typically, if a new kind of message is later introduced for a subclass, it may not be possible to reuse the original code as given so that the new messages can also be handled. The simplicity of the solution provided by Maude, which does not require any explicit code for synchronization, is due to its declarative character and to the very fine granularity of its code, where the basic program units are rewrite rules. The code for a class is therefore an unstructured *set* of rewrite rules, with each rule acting independently of the others. For a subclass, 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. There is no room here for a more detailed discussion of this solution, which can be found in [74], except to mention that cases where the rules of a superclass cannot be used as originally given but must be modified in order to obtain a somewhat different behavior can be handled with similar ease using the module inheritance techniques described in Section 4.3.

4.3 Module Inheritance

There are indeed cases in which one does actually want to *modify* the original code to adapt it to a somewhat different situation. The above class and rule inheritance mechanisms will typically *not* help in such cases. Rather than doing violence to class inheritance and rule inheritance in order to force upon them the job of modifying code, the solution adopted in Maude is to insist on keeping what can be described as an *order-sorted* semantics for class inheritance, and then to provide different *module inheritance* mechanisms to do the job of code modification. An example already discussed in the functional case is the modification of the `LIST` module to obtain the `MSET` module, and other examples illustrating module inheritance by importation, parameterization, renaming or their combination have already been given as well. This distinction between the level of classes (more generally sorts) and the level of modules was already clearly made in the FOOPS language (besides the original paper [49], see also [54] 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 [53].

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` way;
2. *adding new equations or rules* to an imported module;
3. *renaming* some of the sorts or operations of a module;
4. *instantiating* a parameterized module by means of *views*;
5. *adding modules* to form their union;
6. *redefining* an operator so that its syntax and sort requirements are kept intact, but its semantics can be changed by discarding previously given rules or equations involving the operator so that new rules or equations can be then given in their place;

7. *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 [53]. The operations 6–7 are new and permit giving a simple solution to the thorny problem of *message (or method) specialization* without in any way complicating the class inheritance relation, which remains based on an order-sorted semantics. The need for message specialization, i.e., for providing a somewhat different behavior for a message already defined in an old class when received by objects in a new class arises frequently in practice. For example, a bank may at some point want to introduce a new kind of checking accounts in which there is a charge of 50 cents for each cashed check, and then the updating of an account's balance upon receipt of a message of type `(chk A # K amt M)` has to be modified by the extra 50 cents charge. The problem is that if the new class of checking accounts with checking charges is defined as a subclass of the old, then the nice property of rule inheritance derived from subclass inheritance as defined in Section 4.2 is completely destroyed, because the rules for the superclass should *not* be inherited in the new subclass and would in fact produce the wrong behavior.

The solution given to this problem in Maude 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 `CHK-ACCNT` module in the definition of the `CHK(0/.50)ACCNT` module that introduces the new class of checking accounts with charges and also keeps around the previous class of checking accounts without charges.

```
omod CHK(0/.50)ACCNT is
  extending CHK-ACCNT .
  using CHK-ACCNT*(class ChkAccnt to Chk.50Accnt, rdfn(msg chk_#_amt_)) .
  var A : OId .
  vars M N : NNReal .
  var K : Nat .
  var H : ChkHist .
  rl (chk A # K amt M) < A : Chk.50Accnt | bal: N, chk-hist: H >
     => < A : Chk.50Accnt | bal: N - (M + .50), chk-hist: H << K ; M >> >
       if N >= (M + .50) .
endom
```

What the module expression

```
  using CHK-ACCNT*(class ChkAccnt to Chk.50Accnt, rdfn(msg chk_#_amt_)) .
```

indicates is that a new copy of the `CHK-ACCNT` module is created and imported (but without copying its submodules, for example `ACCOUNT`, which are *shared*) in such a way that the class `ChkAccnt` is renamed to `Chk.50Accnt` and the message `(msg chk_#_amt_)` is *redefined*, i.e., its syntax and sort information are maintained, but the rule defining the behavior of the message is discarded. The new behavior is then introduced in the new rule given at the end of the module. Space limitations preclude giving a detailed account of the `rdfn` and `rmv` (remove) commands; this will be done elsewhere. However, we can give a fully detailed account of the use of the `rdfn` command in the above example by presenting a corresponding system module reduction:

```
mod CHK(0/.50)ACCNT# is
  extending CHK-ACCNT# .
  sorts Chk.50Accnt <Chk.50Accnt .
  subsort Chk.50Accnt < Accnt .
  subsort <Chk.50Accnt < <Accnt .
  op Chk.50Accnt : -> <Chk.50Accnt .
  var A : OId .
  var X : <Chk.50Accnt .
  var ATTS : Attributes .
  vars M N : NNReal .
  var K : Nat .
  var H : ChkHist .
  sct < A : X | bal: N, chk-hist: H, ATTS > : Chk.50Accnt .
  rl (chk A # K amt M) < A : X | bal: N, chk-hist: H, ATTS >
    => < A : X | bal: N - (M + .50), chk-hist: H << K ; M >>, ATTS >
       if N >= (M + .50) .
endm
```

The essential point is that, although both `ChkAccnt` and `Chk.50Accnt` are subclasses of `Accnt`, the class `Chk.50Accnt` is *not* a subclass of `ChkAccnt`. Therefore, in the context of the new module, the old rule for messages of the form `(chk A # K amt M)` and the new rule charging 50 cents *coexist without interference*, because they apply to different objects in two different classes that are *incomparable* in the class hierarchy. By contrast, the rules for crediting and debiting accounts and for transferring funds between accounts apply, thanks to rule inheritance, to all classes of accounts, precisely because all of them are subclasses of `Accnt`.

The distinction between class inheritance and module inheritance can be illustrated in this example by means of the diagrams in Figure 7, 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 *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 `CHK-ACCNT` is inherited in *two* different ways by the module `CHK(0/.50)ACCNT`. In this way we can have great flexibility of code reuse *and* a precise and satisfactory 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.

To illustrate the many possibilities for module inheritance that Maude's module operations permit we show below a different, and in fact more general, way of obtaining the two checking account classes in our example. The idea is to define a parameterized module for checking accounts with a checking charge, where the amount of the charge is a parameter, and then to inherit it by two different instantiations to obtain the two desired classes.

```
fth NNREAL* is
  protecting REAL .
```

$$\begin{tikzcd}
 & \text{Accnt} & & & \text{ACCNT} & \\
\text{ChkAccnt} \arrow[ur] & & \text{Chk.50Accnt} \arrow[ul] & \text{CHK-ACCNT} \arrow[ur] \arrow[rr] \arrow[rr, dashed, shift right] & & \text{CHK(0/.50)ACCNT} \arrow[ul]
\end{tikzcd}$$

Figure 7: Class inheritance vs. module inheritance for bank accounts.

```
    op c : -> NNReal .
  endft

  omod CHRG-CHK-ACCNT[C :: NNREAL*] is
    extending ACCNT .
    dfn ChkHist is LIST[2TUPLE[Nat,NNReal]] .
    class ChkAccnt | chk-hist: ChkHist .
    subclass ChkAccnt < Accnt .
    msg chk_#_amt_ : OId Nat NNReal -> Msg .
    var A : OId .
    vars M N : NNReal .
    var K : Nat .
    var H : ChkHist .
    rl (chk A # K amt M) < A : ChkAccnt | bal: N, chk-hist: H >
       => < A : ChkAccnt | bal: N - (M + c), chk-hist: H << K ; M >> >
         if N >= (M + c) .
  endom
```

then we can define the module with our two desired classes as follows,

```
  omod CHK(0/.50)ACCNT is
    extending CHRG-CHK-ACCNT[view to REAL is op c to 0 . endv] .
    extending CHRG-CHK-ACCNT[view to REAL is op c to .50 . endv]*(
                       sort ChkAccnt to Chk.50Accnt) .
  endom
```

4.4 Object Creation and Deletion

In Maude, object creation and deletion can be treated very simply. In fact, a variety of approaches are possible to ensure the key property required, namely that, under appropriate initial conditions, we never reach a configuration in which two different objects have the same name. One possibility is to make object generation *indirect*, in the sense of being mediated by messages of one of the two types below

```
   new(C | ATTS)    (new C | ATTS ack A req R)
```

where the first type of message requests that a new object of class C with attributes ATTS is created, and the second makes the same request but also requires sending an acknowledgement to object `A` with the name of the new object corresponding to its request called `R`. What we must assume is that in the initial configuration we only have a collection of *different* objects in class `ProtoObject` and having a single attribute `counter` whose value is a natural number. Then the rules for creating objects are

```
new(C | ATTS) < O : ProtoObject | counter: N >
   => < O : ProtoObject | counter: N + 1 > < O.N : C | ATTS >

(new C | ATTS ack A req R) < O : ProtoObject | counter: N >
   => < O : ProtoObject | counter: N + 1 > < O.N : C | ATTS >
      (to A : R is O.N)
```

This scheme will of course guarantee that all the names of created objects are different. Since there is in principle no bound on the number of proto-objects in the initial configuration, this scheme can be highly distributed so that object creation does not become a bottleneck. For example, in a multicomputer implementation it would be natural to provide one proto-object per processor in a built-in fashion.

The above scheme for indirect object creation can be slightly modified in order to force some of the attributes of an object to always have fixed initial values (this can be required by means of an `initially` declaration [74] fixing the initial value of an attribute). A more flexible but similar idea is to allow any initial value for an attribute, but to provide a *default* value when the value is not explicitly mentioned in the `new` message; such default values could either be declared explicitly or, as in FOOPS [49], could be automatically computed by the system following some conventions. What is required to support both default and fixed initial attribute values is to make the state of proto-objects richer—so that they contain the appropriate default and `initially` information for the relevant attributes in each class—and to make the above rules for `new` messages conditional to the outcome of checking such information.

A similarly indirect approach to object deletion is to use messages of one of the two types below

```
delete(A)    (delete A ack B)
```

with corresponding rules

```
delete(A) < A : X | ATTS > => null

(delete A ack B) < A : X | ATTS > => (to B : A deleted)
```

We can however allow *direct* generation or deletion of objects in a variety of very general circumstances, including situations in which both direct and indirect schemes are used without conflicting with each other. For example, a very easy scheme for direct object generation that is compatible with the indirect scheme just presented is to assume that one of the objects, call it O, matched by the lefthand side of a rule of the form (†) and surviving in the righthand side of the rule in question has a counter, say with value N. Then, the p new objects created by the righthand side are given names $O.N.1, \ldots, O.N.p$, and O's counter is increased. An example of direct object deletion where objects garbage-collect themselves is given in the eight queens example of Section 6.1.

In summary, there are many ways by which the *uniqueness* of objects can always be guaranteed in an object-oriented system so that the requirement that the rules preserve this uniqueness is satisfied. In addition, object creation can be realized in a highly distributed way. The particular choice of mechanisms and the corresponding choice of data representations for object identifiers may depend on particular characteristics of the given application.

4.5 Broadcasting Messages

Besides the communication that objects can perform among themselves, it is often very desirable to provide more global types of communication so that, for example, all the objects in a given class receive a certain message. In [69] a scheme for such broadcast communication was proposed based on the idea of having a "metaobject" for each class containing the list of all the current objects in the class. This metaobject could broadcast a message to all such objects when appropriate. However, for massively parallel computations involving large numbers of objects distributed across many processors this scheme would be undesirable, although it could be improved by splitting the class metaobject into a hierarchy of metaobjects.

A better approach, developed in joint work with Timothy Winkler and to be further expanded elsewhere, will be briefly sketched below. The key idea is to absorb broadcast communication within the *structural axioms* E of the rewrite theory in question. This means that a broadcast can be viewed at this abstract level as happening "by magic" in the sense that structural axioms are supported by what might be called an "invisible infrastructure" which of course in practice will require some concrete architectural implementation.

Since broadcasting is a global operation, it should be understood as having an entire *configuration* as one of its arguments, for example the configuration making up the global state of all objects and messages at a particular stage of a concurrent computation. The other two arguments can for example be a message and a class identifier `C` indicating the class to which the message should be broadcast. Since the message's addressee is not unique, it is best to assume a generic address denoted by the constant "`*`" that can later be replaced by the real address, so that our message will initially be of the form `M(*)`. In Maude, the keyword `ax` is reserved for equations that are part of the structural axioms E of the rewrite theory specified in a module. Here are the structural axioms for broadcast communication:

```
vars S S' : Configuration .
var MSG : Msg .

ax broadcast(M(*),C,S S') = broadcast(M(*),C,S) broadcast(M(*),C,S') .
ax broadcast(M(*),C,MSG) = MSG .
ax broadcast(M(*),C,< O : D | ATTS >) = if (D : <C)
     then (subst * by O in M(*)) < O : D | ATTS >
     else < O : D | ATTS > fi .
ax broadcast(M(*),C,null) = null .
```

where the predicate `(D : <C)` is a special instance of the notion of a *sort predicate* $t : s$, testing whether a given term t has sort s, that is built-in but can be equationally axiomatized using "retract operators" [51]. The term `(subst * by O in M(*))` corresponds to the application of a substitution operator whose result is `M(O)`, and can be equationally axiomatized for each type of message.

The effect of the above structural axioms is that the broadcast will travel as if by magic across the entire configuration, but its only effect will be that a message is created for each object in the class `C`. This treatment of broadcasting shows the advantage of rewriting logic's great freedom of choice for the appropriate structural axioms of a rewrite theory, which makes possible giving an account of broadcast in purely logical terms. By way of related work, Andreoli and Pareschi [9] consider broadcast in the context of their linear logic based language LO.

4.6 Reflection

The idea of a broadcast suggests the presence of a "master" or "controller" on whose behalf the broadcast is performed. Since such a master has access to an entire configuration, it is natural to conceptualize it as a type of "metaobject" having a configuration as one of its attributes, say

```
    < M : Master | configuration: S, ATTS >
```

that can then initiate a broadcast for the configuration `S` under appropriate conditions specified by rewrite rules for the class `Master`. Rewrite rules for this class may also specify many other actions, such as those involved in detecting global termination, in reacting to various types of feedback from objects in the configuration, or in performing communication with the "outside world." In particular, actions taken by "reflecting" on the global state of the computation seem naturally expressible in this way. All this means that, in general, we should not think of the sort `Value` for the values of attributes as involving only data in sorts of functional modules, but as being capable of involving at times entire configurations, with a subsort relation `Configuration < Value`.

This leads to a more structured view of the concurrent state of an object-oriented system, not as a single configuration, but instead as a possibly quite complex ensemble of configurations and objects that can contain each other like Russian dolls. This view seems closely related the notion of "group objects" proposed by Watari et al. [102], and to Matsuoka et al.'s notion of "hybrid group reflective architecture" [66]. In this way, reflection can be used to better exploit resources at different levels and to respond dynamically to changes in a concurrent computation. Since not one but many "metaobjects" can exist and can even contain other metaobjects, this permits a very distributed style of reflection and allows division of labor, so that higher level metaobjects may only have to deal with simpler and more abstract representations of the distributed state that they control, perhaps indirectly, at lower levels.

For example, besides a master object controlling an entire parallel execution, there can be a *group object* associated with each processor of the parallel machine in which the execution is taking place. Under the control of such a group object there can be a set of ordinary objects for which the group object provides services such as scheduling, sending and delivering messages to and from objects in other processors, etc. For some purposes, including for example broadcasting, the master metaobject may only have to communicate with these group objects, without having to directly involve the objects contained inside each group object. In general, this style of concurrent reflection may involve messages going up and down an entire hierarchy of metaobjects. We can therefore refine more and more our previous view of the master object in the way sketched below

```
 < M : Master | configuration: (< G1: Group | configuration: S1, ATTS1 > S
                                < Gn: Group | configuration: Sn, ATTSn >), ATTS >
```

One way in which rewriting logic seems promising in the context of concurrent object-oriented reflection is that it could provide a simple and uniform semantic basis to deal with

Actors	OOP
Script	Class declaration
Actor	Object
Actor Machine	Object State
Task	Message
Acquaintances	Attributes

Figure 8: A dictionary for Actors.

metaobjects. It appears that they could be dealt with in exactly the same way as ordinary objects, with appropriate rewrite rules specifying the behavior of different metaobject classes such as `Master`, `Group`, etc.

A related topic—discussed also in [69]—is that entities such as classes, modules, etc. can also be represented as metaobjects. This can lead to very flexible and adaptable language implementations using the *metaobject protocol* methodology [58], a methodology that has been adapted to the concurrent case by Matsuoka et al. in [66].

Finding a simple and rigorous semantics for concurrent object-oriented reflection is considered an important open problem, because this area is undergoing rapid development and seems very promising in order to better control the execution of concurrent systems so as to use the computational resources in an optimal way. We refer the reader to [99] for a recent collection of papers on reflection, including object-oriented approaches. In future work we hope to further explore the use of rewriting logic in reflection.

4.7 Actors

Actors [3, 2] provide a flexible and attractive style of concurrent object-oriented programming. However, their mathematical structure, although already described and studied by previous researchers [28, 2], has remained somewhat hard to understand and, as a consequence, the use of formal methods to reason about actor systems has remained limited. The present logical theory of concurrent objects sheds new light on the mathematical structure of actors and provides a new formal basis for the study of this important approach.

Specifically, the general logical theory of concurrent objects presented in this paper directly yields as a special case an entirely declarative approach to the theory and programming practice of actors. The specialization of our model to that of actors can be obtained by first clarifying terminological issues and then studying their definition by Agha and Hewitt [3].

Actor theory has a terminology of its own which, to make things clearer, we will attempt to relate to the more standard terminology employed in object-oriented programming. To the best of our understanding, the table in Figure 8 provides a basic terminological correspondence of this kind.

The essential idea about actors is clearly summarized in the words of Agha and Hewitt [3] as follows:

> “An actor is a computational agent which carries out its actions in response to processing a communication. The actions it may perform are:
>
> - Send communications to itself or to other actors.
> - Create more actors.

- Specify the *replacement behavior*."

The "replacement behavior" is yet another term to describe the new "actor machine" produced after processing the communication, i.e., the new state of the actor.

We can now put all this information together and simply conclude that a logical axiomatization in rewriting logic of an actor system—which is of course at the same time an *executable* specification of such a system in Maude—exactly corresponds to the special case of a concurrent object-oriented system in our sense whose rewrite rules instead of being of the general form (†) are of the special asynchronous and unconditional form

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

Therefore, the present theory is considerably *more general* than that of actors. In comparison with existing accounts about actors [3, 2] it seems also fair to say that our theory is *more abstract* so that some of those accounts might be now regarded as *high level architectural descriptions* of particular ways in which the special case of the abstract model corresponding to actors can be implemented. In particular, the all-important *mail system* used in those accounts to buffer communication is the implementation counterpart of what in our model is abstractly achieved by the *ACI* axioms.

Another nice feature of our approach is that it gives a *truly concurrent* formulation—in terms of concurrent *ACI*-rewriting—of actor computations, which seems most natural given their character. By contrast, Agha [2] presents an interleaving model of sequentialized transitions. Agha is keenly aware of the inadequacy of reducing the essence of true concurrency to nondeterminism and therefore states (pg. 82) that the correspondence between his interleaving model and the truly concurrent computation of actors is "*representationalistic, not metaphysical*."

Yet another contribution of rewriting logic is its simple and unproblematic support for class inheritance in a concurrent context. It seems to be a generally accepted folklore opinion that it is very difficult to support both concurrency and class inheritance in actor languages; in fact, a number of actor languages do not support inheritance for this reason, and the whole matter is considered an open problem for which only partial or tentative solutions seem to have been suggested so far. Maude's simple solution to this "inheritance anomaly" [67, 90] was already mentioned in Section 4.2 and is discussed in full detail in [74].

It is also important to point out that, in our account, the way in which an object changes its state as a consequence of receiving a message may involve many concurrent rewritings of its attributes, i.e., objects exhibit *intra-object concurrency*; by contrast, typical actor languages treat change of object state as a sequential computation, and formal models of concurrency for actors such as that in [2] only deal with message-passing inter-object concurrency. In this sense, the concurrent rewriting model is considerably more fine grained, and when implemented on an appropriate architecture such as the RRM (see Section 5.3) directly supports the massive exploitation of both inter-object parallelism (typically by interprocessor communication) and intra-object parallelism (typically by SIMD data parallelism). A good example exhibiting these two types of parallelism is the search for solutions to the eight queens problem in Section 6.1.

figure=/homes/dumas/winkler/text/fig/bign3.ps,scale=100

Figure 9: Unification of concurrency models.

There is one additional aspect important for actor systems and in general for concurrent systems, namely *fairness*. For actors, this takes the form of requiring *guarantee of mail delivery*. The issue of fairness for term rewriting has already received attention by several authors, including Francez and Porat [92] and Tison [100]. Indeed, it is possible to state precisely a variety of fairness conditions for concurrent rewriting; in particular, one could express the guarantee of mail delivery for the special case of actors in these terms. However, a detailed treatment of this topic is outside the scope of this paper and will have to wait for a future occasion.

4.8 Generality of the Concurrent Rewriting Model

Concurrent rewriting is a very general model of concurrency from which many other models—besides those discussed in this paper—can be obtained by specialization. Space limitations preclude a detailed discussion, for which we refer the reader to [72]. However, we can summarize such specializations using Figure 9, where CR stands for concurrent rewriting, the arrows indicate specializations, and the subscripts $\emptyset$, AI, and ACI stand for syntactic rewriting, rewriting modulo associativity and identity, and ACI-rewriting respectively. Within syntactic rewriting we have labelled transition systems, which are used in interleaving approaches to concurrency; functional programming (in particular Maude's functional modules) corresponds to the case of *confluent*[19] rules, and includes the λ-calculus (see Section 5.2) and the Herbrand-Gödel-Kleene theory of recursive functions. Rewriting modulo AI yields Post systems and related grammar formalisms, including Turing machines. Besides the general treatment by ACI-rewriting of concurrent object-oriented programming that contains Actors as a special case [3], rewriting modulo ACI includes Petri nets [93], the Gamma language of Banâtre and Le Mètayer [13], and Berry and Boudol's *chemical abstract machine* [15] (which itself specializes to CCS [84]; see [15]), as well as Unity's model of computation [26]; another special case is Engelfriet et al.'s POPs and POTs which are higher level Petri nets for actors [36, 35].

The ACI case is quite important, since it contains as special subcases a good number of concurrency models that have already been studied. In fact, the associativity and commutativity of the axioms appear in some of those models as "fundamental laws of concurrency." However, from the perspective of this work the ACI case—while being important and useful—does not have a monopoly on the concurrency business. Indeed, "fundamental laws of concurrency" expressing associativity and commutativity are only valid in this particular case. They are for example meaningless for the tree-structured case of functional programming. The point is that the laws satisfied by a concurrent system cannot be determined *a priori*. They essentially depend on the actual distributed structure of the system, which is its algebraic structure.

More importantly—and this is a key advantage of Maude's object-oriented modules—an ACI operator, even when present, does account only for *some* of the concurrency of a system when other operators not having that property are also present. For example, an object may communicate with other objects in an ACI distributed state, but its attributes can also have a distributed structure—typically a tree structure—so that their updating can be performed in

[19]Although not reflected in the picture, rules confluent *modulo* equations E are also functional.

parallel. Concurrent rewriting does not discriminate between one level of parallelism (*ACI* communication between objects) and another (parallel attribute updating); instead, it integrates both levels within the same formal framework supporting concurrency *at all levels*.

Of course, the general claim that a system's distributed structure coincides with its algebraic structure applies also here. The *ACI* axioms lead to a state structure that is distributed as a *commutative word*, *multiset*, or *bag*, all these being different expressions for the same idea. This is a very fluid and flexible structure which, in particular, is an ideal abstract structure for *communication*; as already pointed out, this may only account for the *top-level structure* of a system, which in the framework of rewriting logic is seamlessly integrated with any other distributed structures at lower levels.

5 Simple Maude

This section summarizes joint work with Timothy Winkler [83] on the design of a sublanguage of Maude called Simple Maude chosen with the purpose of being implementable with reasonable efficiency on a wide variety of parallel machines. The present summary will focus primarily on Simple Maude and will touch more briefly on implementation issues and on the use of Maude as glue to parallelize conventional programs and to put together heterogeneous systems; more details are given in the joint paper [83].

5.1 Maude as a Wide Spectrum Language

Although concurrent rewriting is a general and flexible model of concurrency and can certainly be used to reason formally about concurrent systems at a high level of abstraction, it would not be reasonable to implement this model for programming purposes in its fullest generality. This is due to the fact that, in its most general form, rewriting can take place *modulo* an arbitrary equational theory E which could be undecidable. Of course, a minimum practical requirement for E is the existence of an algorithm for finding all the matches modulo E for a given rule and term; however, for some axioms E, this process, even if it is available, can be quite inefficient, so that its implementation should be considered a theorem proving matter, or at best something to be supported by an implementation for uses such as rapid prototyping and execution of specifications, but probably should not be made part of a programming language implementation. A good example is general AC-rewriting, which can be quite costly for complicated lefthand side patterns; this can be acceptable for rapid prototyping purposes—in fact, the OBJ3 interpreter [45, 53] supports this as well as rewriting modulo other similar sets of axioms E—but seems to us impractical for programming purposes even if a parallel implementation is considered[20].

In this regard, it is useful to adopt a *transformational* point of view. For specification purposes we can allow the full generality of the concurrent rewriting model, whereas for programming purposes we should study subcases that can be efficiently implemented; executable specifications hold a middle ground in which we can be considerably more tolerant of inefficiencies in exchange for a greater expressiveness. The idea is then to develop program transformation techniques that are semantics-preserving and move us from specifications to programs, and from less efficient programs—perhaps just executable specifications—to more efficient ones. This transformational approach fits in very nicely with the design of Maude which, as with OBJ3 in

[20]Of course, even in a case like this there can be different opinions. Banâtre, Coutant, and Le Mètayer have in fact considered parallel machine implementations of AC-rewriting for their Gamma language [12].

Rewriting Logic: Specification

Maude:
Prototyping and
Executable Specification, Debugging

Simple Maude:
Parallel Programming

Figure 10: Maude and Simple Maude as subsets of Rewriting Logic.

the functional case, can be regarded as a wide spectrum language that integrates both specification and computation. Indeed, Maude *theories*, whether functional, system, or object-oriented, are used for specification purposes—for example, to specify the semantic requirements of the parameters of a parameterized module—and therefore need not be executable[21]. For Maude *modules*, which are of course executable, a distinction should be made between use for rapid prototyping and executable specification, and use for programming, with more stringent restrictions imposed in the latter case.

This suggests considering two subsets of rewriting logic. The first subset gives rise to Maude—in the sense that Maude modules are rewriting logic theories in that subset—and 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 programming purposes for which a wide variety of machine implementations can be developed. Program transformation techniques can then support passage from general rewrite theories to Maude modules and from them to modules in Simple Maude. Figure 10 summarizes the three levels involved.

Regarding Maude and its implementation as an interpreter, we plan to support rewriting modulo all the axioms supported by OBJ3, where a binary operator can be declared to be associative and/or commutative and/or having a neutral element, and rewriting modulo a combination of those axioms is supported by the implementation. In particular, all the modules in this paper are executable Maude modules. The Maude interpreter will support rapid prototyping and debugging of system designs and specifications that, if desired, could also be used to derive an efficient system by applying to them a series of semantics-preserving transformations and refinement steps bringing the entire program within the Simple Maude sublanguage; some program transformations can be automated so that a user can write certain types of programs in a more abstract way in Maude and could leave the task of transforming them into Simple Maude programs to a compiler.

[21]In fact, for the sake of greater expressiveness they may even be theories in logics different from rewriting or equational logic; details will appear elsewhere.

5.2 Simple Maude

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. As discussed briefly later, Simple Maude can also support multilingual extensions and can be used as the glue for putting together open heterogeneous systems encompassing many different machines and special I/O devices. This section summarizes the language conventions for functional, system, and object-oriented modules in Simple Maude.

5.2.1 Rewriting Modulo Church-Rosser and Terminating Equations

As work on compilation techniques for functional languages has amply demonstrated, syntactic rewriting, i.e., rewriting modulo an empty set of structural axioms, can be implemented efficiently on sequential machines; our experience with parallel compilation techniques for syntactic rewriting in the Rewrite Rule Machine (RRM) project [50, 6, 5] leads us to believe that this can be done even more efficiently on parallel machines. Therefore, functional or system modules with an empty set of structural axioms are among the easiest to implement and belong to Simple Maude.

A closely related class of modules also allowed in Simple Maude is that of functional or system modules having an associated rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ such that the set E of structural axioms is Church-Rosser and terminating and has the additional property that, for the rewrite theory $\mathcal{R}' = (\Sigma, \emptyset, L, R)$, whenever we have $\mathcal{R}' \vdash t \longrightarrow t'$ we also have $\mathcal{R}' \vdash can_E(t) \longrightarrow t''$ with $can_E(t') = can_E(t'')$, where $can_E(t)$ denotes the (canonical) normal form to which the term t is reduced by the equations E used as rewrite rules. Under such circumstances we can implement rewriting modulo E by syntactic rewriting with the rewrite theory $\mathcal{R}'' = (\Sigma, \emptyset, L, R \cup E)$ provided that we restrict our attention to sequents of the form $can_E(t) \longrightarrow can_E(t')$ which faithfully simulate $\mathcal{R}$-rewritings in E-equivalence classes. For modules of this kind the structural axioms $u = v$ in E are introduced by the syntax

```
ax u = v .
```

A functional module defined by a rewrite theory $\mathcal{R}$ of this kind has the same initial algebra as the functional module defined by the associated rewrite theory $\mathcal{R}''$. However, the semantics of both modules as defined in Section 8 are different. For system modules of this kind, the semantics associated with $\mathcal{R}$ and with the rewrite theory $\mathcal{R}''$ that simulates it are even more different.

Even for functional modules, the possibility of allowing a distinction between rules and structural axioms is quite convenient and meaningful. For example, we can in this way avoid all the fuss with variables and substitution in the standard lambda calculus notation by defining a functional module `LAMBDA` corresponding to the $\lambda\sigma$-calculus of Abadi, Cardelli, Curien, and Lévy [1] in which we interpret their *Beta* rule as the only rule, and their set σ of Church-Rosser and terminating equations for explicit substitution as the set E of structural axioms. The point is that σ-equivalence classes are isomorphic to standard lambda expressions (modulo α conversion), and rewritings in σ-equivalence classes correspond to β-reductions.

There are of course a variety of strategies by which we can interleave E-rewriting and R-rewriting in the implementation of modules of this kind, and some strategies can be more efficient than others. It could also be possible to generalize the class of modules that can be implemented by syntactic rewriting by giving weaker conditions on the relationship between R and E that are still correct assuming a particular strategy for interleaving R- and E-rewritings.

5.2.2 Object-Oriented Modules

In Maude, the essence of concurrent object-oriented computation is captured by concurrent ACI-rewriting using rules of the general form (†) described in Section 4.1.1. In a sequential interpreter this can be *simulated* by performing ACI-rewriting using an ACI-matching algorithm. However, in a parallel implementation ACI-rewriting should be realized exclusively by means of *communication*. The problem is that realizing general AC- or ACI-rewriting in this way can require unacceptable amounts of communication and therefore can be very inefficient, even for rules of the form (†) introduced in object-oriented modules that make a somewhat limited use of ACI-rewriting because no variables are ever used to match multisets. For this reason, our approach to the object-oriented modules of Simple Maude is to only allow conditional rules of the form

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

involving only one object and one message in their lefthand side, where $p, q \geq 0$, and where the notation $(\langle O : F' \mid atts' \rangle)$ 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. We allow as well conditional rules for autonomous objects of the form (again, with $p, q \geq 0$)

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

Specifically, the lefthand sides in rules of the form (‡) should fit the general pattern[22]

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

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 AC-rewriting by communication is available to us for rules of the form (‡), 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. For example, a rule to credit money to an account can be implemented this way by routing the credit message to

[22]However, the rules for object creation using proto-objects given in Section 4.4 which do not fit this pattern are also allowed in Simple Maude.

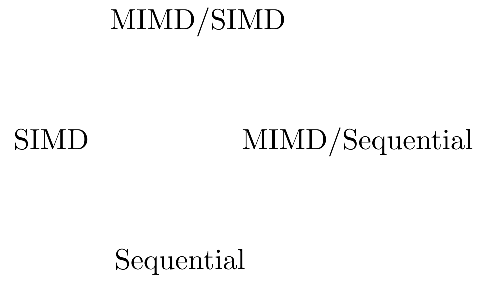

Figure 11: Specialization relationships among parallel architectures.

the location of its addressee so that when both come into contact the rewrite rule for crediting the account can be applied. Rules of the form (§) are even simpler to implement, since their matching does not require any communication by messages; however, both types of rules assume the existence of a basic mechanism for sending the messages generated in the righthand side to their appropriate destination.

How should the gap between the more general rules (†) allowed in Maude modules and the more restricted rules (‡) and (§) permitted in Simple Maude modules be mediated? Our approach to this problem—in forthcoming joint work with Patrick Lincoln and Timothy Winkler—has been to develop program transformation techniques that, under appropriate fairness assumptions, guarantee that rewriting with rules of the form (†) can be simulated by rewriting using rules of the form (‡) and (§). The basic idea is that a (†) rule in general requires the synchronization of several objects—what in some contexts is called a *multiparty interaction*—but this synchronization can be achieved in an asynchronous way by an appropriate sending of messages using rules of the form (‡). Transformations of this kind can be automated and relegated to a compiler, so that a user could write object-oriented modules in Maude and not have to worry about the corresponding expression of his program in Simple Maude. However, Simple Maude is already quite expressive—in particular, more expressive than Actors—and many programs will fall naturally within this class without any need for further transformations.

The strategy described in this section for Simple Maude's object-oriented modules can be generalized to system modules[23] so that they can also perform concurrent AC-rewriting by asynchronous message-passing communication; this generalization is discussed in [83].

5.3 Sequential, SIMD, MIMD, and MIMD/SIMD Implementations

Simple Maude can be implemented on a wide variety of parallel architectures. The diagram in Figure 11 shows the relationship among some general classes that we have considered. There are two orthogonal choices giving rise to four classes of machines: the processing nodes can be either a single sequential processor or a SIMD array of processors, and there can be either just a single processing node or a network of them. The arrows in the diamond denote specializations from a more general and concurrent architecture to degenerate special cases, with the sequential case at the bottom. The arrows pointing to the left correspond to specializing a network of

[23]As discussed before, we also allow Simple Maude system modules where the structural axioms E are given by Church-Rosser and terminating equations.

processing nodes to the degenerate case with only one processing node; the arrows pointing to the right correspond to specializing a SIMD array to the degenerate case of a single processor.

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 four classes of machines; the paper [83] discusses this matter in greater detail; here we limit ourselves to a brief sketch. In the MIMD/Sequential (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 sequential case corresponds to a single conventional sequential processor and can be viewed as the degenerate case of a MIMD/Sequential machine with only one processor. In this case, at most one rule is applied to a single place in the data at a time. Since there is only one processor in which all objects are located, implementing rules of the form (‡) does not require any interprocessor communication. 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) [52, 7, 6, 5] 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/Sequential 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.

5.4 Multilingual Extensions and Open Heterogeneous Systems

Simple Maude will support the integration of modules written in conventional languages such as Fortran and C; this will allow reusing and parallelizing code originally developed for sequential machines. More generally, not only conventional code, but entire systems and special-purpose hardware devices can be integrated in a similar way. The joint paper [83] provides a more detailed discussion of this aspect of the language, which is briefly summarized below.

The way in which this integration can be accomplished generalizes a facility already available in Maude's functional sublanguage (OBJ) for defining *built-in sorts* and *built-in rules* [45, 53]. This facility has provided valuable experience with multilingual support, in this case for OBJ and Common Lisp, and can be generalized to a facility for defining *foreign interface modules* in Maude. Such foreign interface modules have abstract interfaces that allow them to be integrated with other Maude modules and to be executed concurrently with other computations; however, they are treated as "black boxes." In particular, Maude's model of computation and its modular style provide a simple way of gluing a concurrent program together out of pieces that can be either written in Simple Maude or can be sequential code written in conventional languages. Foreign interface modules may provide either a functional data type, or an object-oriented class. In the first case, the treatment will be extremely similar to that provided in OBJ. In the second case, the abstract interface will be provided by the specification of the messages that act upon the new class of objects. This second case is also the approach used to interface to existing systems or applications and to special-purpose hardware devices; they are treated as, possibly quite complex, black boxes.

Future computing environments will be heterogeneous, involving many different machine architectures, and evolving, with components being attached and removed over time. It has long been recognized that message-passing models provide one of the best ways to integrate distributed heterogeneous systems. Designs based on message passing also make it relatively easy to add or remove resources, and to deal with variations in the size of parallel architectures (the number of processing nodes). The advantage of the concurrent rewriting model is that it integrates message passing within a simple and mathematically precise abstract model of computation and can therefore be used as both a semantic framework and as a language for the integration of heterogeneous systems. In order to incorporate something (whether it be a special-purpose processor, an I/O device, a database, or a program written in C) into the abstract model, it is just necessary to treat it as a black box and to specify its interface to the system, i.e., the message protocol that is used for interacting with it.

Related efforts in multilingual support for parallel programming include: the *Linda* language developed by D. Gelernter and his collaborators at Yale [25], the *Strand* language designed by I. Foster and S. Taylor [37], the *Program Composition Notation* (PCN) designed by K. M. Chandy and S. Taylor at Caltech [27], and the *GLU* language developed by R. Jagannathan and A. Faustini at SRI International [57].

6 More Examples

This section presents two somewhat longer Maude examples: a module for autonomous objects that search all the solutions to the eight queens problem in parallel, and a fault-tolerant communication protocol.

6.1 Eight Queens

The following example illustrates several Maude features, including autonomous objects, object creation and deletion, the convenience of the notational conventions for rules involving objects introduced in Section 4.2 that allow omitting mention of attributes irrelevant for a particular rule, extensive use of intra-object concurrency, and parameterization. The example is the well-known eight queens problem of finding all board configurations on an 8×8 board in which eight queens are placed on the board in such a way that no queen is able to capture any other queen. According to the rules of chess, a queen can capture any other piece along its same row, column, or diagonals. An actor treatment of this problem has appeared in [11]. The idea in both [11] and in the example below is to search all the solutions in parallel without any backtracking.

We make the size of the board a parameter, and represent a *partial solution* as a list of pairs of nonzero natural numbers $(1, n_1), \ldots, (i, n_i)$, indicating that i queens have already been placed successfully on the first i columns of the board, in the positions contained in the list. Of course, since only one queen can be placed per row (or per column) in a solution, the only rows that remain free as potential candidates for placing new queens are in the set $\{1, \ldots, k\} - \{n_1, \ldots, n_i\}$, with $k \times k$ the assumed size of the board. This is the set of row positions in column $i + 1$ that we have to examine to try to expand the partial solution $(1, n_1), \ldots, (i, n_i)$ by adding a new queen in column $i+1$. Since, by construction, there will be only one queen per row and per column in this expanded partial solution, the only thing that we have to check is that the row position chosen in the set of free rows to place the queen on the $i+1$-th column is not in the same diagonal as any of the previous queens. This can be done by applying the predicate `in-diag` defined in the auxiliary functional module `DIAG` below; we can then gather together all successful row positions failing the `in-diag` test by means of the function `good-rows`. We assume that the `INT` module has a sort `NzNat` of nonzero natural numbers, and an absolute value function `|_|`.

```
fmod DIAG is
  protecting INT .
  protecting BOOL .
  protecting LIST[2TUPLE[NzNat,NzNat]*(sort 2Tuple to NzNatPair)]*(
               sort List to PairList) .
  protecting LIST[NzNat]*(sort List to NzNatList) .

  op in-diag : NzNat NzNat PairList -> Bool .

  *** the arguments to in-diag are: a row position, the column being
  *** considered, and a partial solution up to the previous column

  vars C R C' R' : NzNat .
  var P : NzNatPair .
  var L : NzNatList .
  var Q : PairList .

  eq in-diag(R,C,P Q) = in-diag(R,C,P) or in-diag(R,C,Q)
  eq in-diag(R,C,<< C' ; R' >>) = (| R - R' | == | C - C' |) .
  eq in-diag(R,C,nil) = false .

  op good-rows : NzNatList NzNat PairList -> NzNatList .

  *** the arguments to good-rows are: a list of free rows, the column
  *** being considered, and a partial solution up to the previous column

  eq good-rows(R L,C,Q) = good-rows(R,C,Q) good-rows(L,C,Q) .
  eq good-rows(nil,C,Q) = nil .
  eq good-rows(R,C,Q) = if in-diag(R,C,Q) then nil else R fi .
endfm
```

The search for all the solutions will use objects in a class `QAgent` having attributes `caller` (the name of the external object that requested the eight queens solutions), `psol` (a partial solution), `column` (the last column in the partial solution), `free-rows` (the unoccupied rows), `good-rows` (the rows in the next column that can expand the partial solution), and `tested` (whether or not the free rows have been tested as expansions of the solution). The search will begin in a configuration involving the set of k objects

```
< N : QAgent | caller: B, psol: << 1 ; N >>, column: 1,
  free-rows: remove N from (1 ... K), good-rows: nil, tested: false >
```

for N varying from 1 to k, with object N beginning the search at row N in the first column and searching for good rows in column 2. The module defining QAgent has the functional theory NAT* of Section 4.1 as its parameter, to specify the side size k of the $k \times k$ board.

```
omod K-QUEENS[K :: NAT*] is
  protecting DIAG .
  class QAgent | caller: OId, psol: PairList, column: NzNat ,
           free-rows: NzNatList, good-rows: NzNatList, tested: Bool .
  msg to_sol_ : OId PairList -> Msg .
  vars A B : OId .
  vars R C : Nznat .
  var Q : PairList .
  vars L L' : NzNatList .

  *** the agent tests which free rows are good for placing a queen in the
  *** next column, and records the information

  rl < A : QAgent | psol: Q, column: C, free-rows: L,
         tested: false > =>
       < A : QAgent | good-rows: good-rows(L,C + 1,Q), tested: true > .

  *** for each good row found, a new object continuing the search
  *** with that row in the expanded partial solution is created, except
  *** in case the good row is in the last column, where the solution
  *** is sent to the caller and the agent garbage-collects itself

  rl < A : QAgent | caller: B, psol: Q, column: C, free-rows: L,
         good-rows: N L' > =>
     if (C + 1) < k then
       < A : QAgent | good-rows: L' >
       new(QAgent | caller: B, psol: Q << C + 1 ; N >>, column: C + 1,
         free-rows: remove N from L, good-rows: nil, tested: false)
     else (to B sol Q << k ; N >>) fi .

  *** if no good row is left, the agent garbage-collects itself

  rl < A : QAgent | good-rows: nil, tested: true > => null .
endom
```

The classical eight queens case can be obtained by a simple instantiation

```
make 8-QUEENS is K-QUEENS[view to NAT is op k to 8 . endv] endmk
```

6.2 A Communication Protocol Example

If a communication mechanism does not provide reliable, in-order delivery of messages, it may be necessary to generate this service using the given unreliable basis. The following example, developed jointly with Timothy Winkler and borrowed with slight modifications from [83], shows how this might be done. Since unreliable communication is a more serious issue across different machines, this example illustrates the application of Maude to heterogeneous open systems (see Section 5.4). This was derived from the alternating bit protocol as presented in Lam and Shankar [60], although, since we do not assume in-order delivery, we cannot use the alternating bit protocol. The same kind of example is discussed in a somewhat different way in Chandy and Misra [26]. The following definition creates a generic, fault-tolerant connection between a specific sender and receiver pair. Notice that—thanks to the abstractness of the concurrent rewriting model and to the parameterization mechanisms of the language—the module is very general in several respects:

- it makes very few assumptions about the communication between sender and receiver;
- the parameter `ELT` can be instantiated to any type of data to be sent;
- the parameters `S` and `R` can be instantiated to any two previously defined classes of objects.

The requirement that the parameters `S` and `R` have to satisfy is expressed by the object-oriented theory `CLASS` below, whose `Cl` sort can be instantiated to any class of an object-oriented module. To disambiguate the use of the parameter sort `Cl` in `S` and `R`, we use the notation `Cl.S` and `Cl.R`.

```
oth CLASS is
  class Cl .
endoth

omod PROTOCOL[ELT :: TRIV, S :: CLASS, R :: CLASS] is
  protecting LIST[ELT] .
  protecting NAT .

  sort Contents .
  subsort Elt < Contents .
  op empty : -> Contents .
  msg to:_(_,_) : OId Elt Nat -> Msg . *** data to receiver
  msg to:_ack_ : OId Nat -> Msg . *** acknowledgement to sender

  class Sender | rec: OId, sendq: List, sendbuff: Contents ,
                 sendcnt: Nat .
  subclass Sender < Cl.S .

  *** rec is the receiver, sendq is the outgoing queue, sendbuff
  *** is either empty of the current data, sendcnt is the sender
  *** sequence number

  vars S R : OId .
  var N : Nat .
  var E : Elt .
  var L : List .
```

```
    var C : Contents .

    rl produce :
       < S : Sender | rec: R, sendq: E . L,
          sendbuff: empty, sendcnt: N > =>
       < S : Sender | rec: R, sendq: L, sendbuff: E, sendcnt: N + 1 > .

    rl send :
       < S : Sender | rec: R, sendq: L, sendbuff: E, sendcnt: N > =>
       < S : Sender | rec: R, sendq: L, sendbuff: E, sendcnt: N >
       (to: R (E,N)) .

    rl rec-ack :
       < S : Sender | rec: R, sendq: L, sendbuff: C, sendcnt: N >
       (to: S ack M) =>
       < S : Sender | rec: R, sendq: L,
                 sendbuff: (if N == M then empty else C fi),
                 sendcnt: N > .

    class Receiver | sender: OId, recq: List, reccnt: Nat .
    subclass Receiver < Cl.R .

    *** sender is the sender, recq is the incoming queue,
    *** and reccnt is the receiver sequence number

    rl receive :
       < R : Receiver | sender: S, recq: L, reccnt: M > (to: R (E,N)) =>
       (if N == M + 1 then
          < R : Receiver | sender: S, recq: L . E, reccnt: M + 1 >
        else
          < R : Receiver | sender: S, recq: L, reccnt: M >
        fi)
       (to: S ack N)
  endom
```

Under reasonable fairness assumptions, these definitions will generate a reliable, in-order communication mechanism from an unreliable one. The message counts are used to ignore all out-of-order messages. The fairness assumption will ensure that the `send` action and corresponding `receive` actions will be repeated until a `rec-ack` can be performed; thus each `produce` necessarily leads to a corresponding `rec-ack`. Note that the `send` operation is enabled until the corresponding `rec-ack` occurs.

We can explicitly model the fault modes of the communication channel as in the following definition:

```
  omod PROTOCOL-IN-FAULTY-ENV[ELT :: TRIV, S :: CLASS, R :: CLASS] is
     extending PROTOCOL[ELT,S,R] .
```

```
    var M : Msg .

    rl duplicate :
       M => M M .

    class Destroyer | sender: OId, rec: OId, cnt: Nat .

    var N : Nat .
    var E : Elt .
    vars S R D : OId .

    rl destroy1 :
       < D : Destroyer | sender: S, rec: R, cnt: N > (to: R (E,N)) =>
       < D : Destroyer | sender: S, rec: R, cnt: N > .

    rl destroy2 :
       < D : Destroyer | sender: S, rec: R, cnt: N > (to: S ack N) =>
       < D : Destroyer | sender: S, rec: R, cnt: N > .

    rl limited-injury :
       < D : Destroyer | sender: S, receiver: R, cnt: N > =>
       < D : Destroyer | sender: S, receiver: R, cnt: N + 1 > .
 endom
```

Messages may be duplicated or destroyed. The `limited-injury` rule, under an assumption of fair application of the total set of rules, will ensure that messages are not always destroyed. The new system, with the `Destroyer`, will also satisfy the same correctness condition regardless of messages being duplicated, being destroyed, or arriving out of order.

7 Maude and MaudeLog as Multiparadigm Logic Programming Languages

The statement made in the Introduction about Maude being a language directly based on a logic—namely rewriting logic—so that programs in Maude are logical theories and concurrent Maude computation is logical deduction has by now been explained and illustrated with examples in great detail. Calling Maude a "logic programming language" is therefore entirely justified, provided that we use this term in a broad sense that allows choosing among many logics the one that best suits our particular needs. In addition, Section 8 will discuss how initial models for rewriting logic can be used to give a denotational semantics to Maude programs in agreement with the logic programming nature of Maude.

In fact, although this has been kept implicit in the exposition, Maude's design is based on a general axiomatic notion of "logic programming language" which is itself based on a general axiomatic theory of logics. This theory of "general logics" and the associated general notion of "logic programming language" were developed in [68], and were inspired by previous work of Goguen and Burstall on "institutions" [44]. The paper [73] introduces these general concepts, discusses general methods for designing multiparadigm logic programming languages using such concepts, and explains how Maude and MaudeLog were designed according to those methods. We briefly sketch here some of the ideas of [73] to give the reader a better insight about the multiparadigm aspects of Maude and of MaudeLog.

For designing multiparadigm logic programming languages, a key technical tool is the use of *mappings between logics* that relate the syntax, sentences, entailments, and models of two different logics by appropriate translations. Technically, a unification of paradigms is achieved by mapping the logics of each paradigm into a richer logic in which the paradigms are unified.

In the case of Maude and MaudeLog, what is done is to define a new logic—rewriting logic—in which concurrent computations—and in particular concurrent object-oriented computations—can be expressed in a natural way, and then to formally relate this logic to the logics of the functional and relational paradigms, i.e., to equational logic and to Horn logic, by means of maps of logics that provide a simple and rigorous unification of paradigms. As it has already been mentioned, we actually assume an order-sorted structure throughout, and therefore the logics in question are: order-sorted rewriting logic, denoted *OSRWLogic*, order-sorted equational logic, denoted *OSEqtl*, and order-sorted Horn logic, denoted *OSHorn*.

At first sight one might conjecture that in order to extend functional programming to handle concurrent systems and object-oriented computations one has to *complicate* the logic, by embedding equational logic inside a more involved formalism. Maude's solution to this problem, provided by rewriting logic, achieves a very simple unification of functional programming within the broader context of concurrent systems programming and concurrent object-oriented programming by doing *just the opposite*. That is, rewriting logic actually has *simpler* rules of deduction than equational logic; indeed, as has already been mentioned, it can be obtained from equational logic by dropping the symmetry rule. The point, however, is that rewriting logic has a much broader class of models than equational logic (see Section 8), and the new models, corresponding to concurrent systems, are just what we need.

Rather than trying to force nonfunctional applications within a functional world, what it is done in this solution is to abandon any such attempts altogether, i.e., to leave the functional world untouched, and then to show that the logic of functional programming—in the particular variant discussed here of order-sorted equational logic—can be embedded within (order-sorted) rewriting logic by means of a map of logics

$$\mathit{OSEqtl} \longrightarrow \mathit{OSRWLogic}.$$

The details of this map of logics are discussed in Appendix B of [73]. At the programming language level, such a map corresponds to the inclusion of Maude's *functional modules* (essentially identical to OBJ3 modules) within the language. The key point of having a map of this kind is that it relates both the proof-theoretic and the model-theoretic aspects of both logics. This permits maintaining intact the standard *initial algebra semantics* as the mathematical semantics of Maude's functional modules, just as in OBJ3, while allowing for a different semantics—based on the model theory of rewriting logic (see Section 8)—for Maude's system modules and object-oriented modules.

Since the power and the range of applications of a multiparadigm logic programming language can be substantially increased if it is possible to solve queries involving *logical variables* in the sense of relational programming, as in the Prolog language, we are naturally led to seek a unification of the three paradigms of functional, relational and concurrent object-oriented programming into a single multiparadigm logic programming language. This unification can be attained in a language extension of Maude called MaudeLog.

As before, the method used to achieve a unification of this kind in a simple and rigorous manner is to combine the logics involved by means of mappings. In addition to the mapping from equational logic to rewriting logic just discussed, which provides the integration of the functional facet, what remains is the integration of Horn logic. Such an integration is achieved by a map of logics

$$\mathit{OSHorn} \longrightarrow \mathit{OSRWLogic}$$

that systematically relates order-sorted Horn logic to order-sorted rewriting logic. The details of this map are discussed in Appendix C of [73].

The difference between Maude and MaudeLog does not consist in any change in the underlying logic; indeed, both languages are based on rewriting logic, and both have rewrite theories as programs. It resides, rather, in an enlargement of the set of *queries* that can be presented, so that, while keeping the same syntax and models, in MaudeLog we also consider queries involving existential formulas of the form

$$\exists \overline{x} \quad [u_1(\overline{x})] \longrightarrow [v_1(\overline{x})] \wedge \ldots \wedge [u_k(\overline{x})] \longrightarrow [v_k(\overline{x})].$$

Therefore, the sentences and the deductive rules and mechanisms that are now needed require further extensions of rewriting logic deduction. In particular, solving such existential queries requires performing *unification*, specifically—given Maude's typing structure—order-sorted unification [79].

The above map of logics means that, after a simple translation, we can view a (pure) Prolog program as a MaudeLog program. We illustrate this translation by means of a simple example of family relations.

Here is the example in terms of Horn clauses:

```
Parent(X,Y) :- Father(X,Y)
Parent(X,Y) :- Mother(X,Y)
Grandparent(X,Z) :- Parent(X,Y), Parent(Y,Z)
Father(Peter,Paul)
Mother(Mary,Paul)
Father(Arthur,Peter)
Mother(Claire,Peter)
Father(Robert,Mary)
Mother(Louise,Mary)
```

This is the MaudeLog translation[24]:

```
mod FAMILY is
  extending PROP .
  sort People .
  ops father mother parent grandparent : People People -> Prop .
  ops peter paul mary arthur claire robert louise : -> People .
  vars X Y Z : People .
  rl father(X,Y) => parent(X,Y) .
  rl mother(X,Y) => parent(X,Y) .
```

[24]Although the clauses are translated as illustrated below, the rewrite theory obtained by this translation makes additional requirements on the models, as explained in Appendix C of [73].

```
  rl parent(X,Y), parent(Y,Z) => grandparent(X,Z) .
  rl true => father(peter,paul) .
  rl true => mother(mary,paul) .
  rl true => father(arthur,peter) .
  rl true => mother(claire,peter) .
  rl true => father(robert,mary) .
  rl true => mother(louise,mary) .
endm
```

The imported module `PROP` has a conjunction operator `_,_: Prop Prop -> Prop` as a multiset operator (i.e., it is associative and commutative) with identity `true`. In addition, it has the rule `rl X => true`.

Then the query

```
grandparent(X,Paul)?
```

is translated into the existential formula

```
∃X : People .  true => grandparent(X,Paul)
```

To solve this, we do backward search from the goal using the rewrite rules. In fact, because there exists an endomorphism of rewriting logic mapping the theory `FAMILY` to a version with all `=>`'s reversed, this could be reformulated in terms appropriate for forward search if desired.

Although the basic relationship formalized by the map of logics between Horn logic and rewriting logic is well understood, the design of MaudeLog is at a more preliminary stage than that of Maude and much work remains to be done at the theoretical level, in the deduction and operational semantics aspects, at the language design level, and eventually in an actual implementation.

8 Semantics

In this section we discuss models for rewriting logic and explain how such models are used to give semantics to modules in Maude. We will focus on the basic ideas and intuitions and leave out some of the details, which can be found in [72].

8.1 The Models of Rewriting Logic

We first sketch the construction of initial and free models for a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$. Such models capture nicely the intuitive idea of a "rewrite system" in the sense that they are systems whose states are E-equivalence classes of terms, and whose transitions are concurrent rewritings using the rules in R. Such systems have a natural *category* structure [62], with states as objects, transitions as morphisms, and sequential composition as morphism composition, and in them dynamic behavior exactly corresponds to deduction.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, the model that we are seeking is a category $\mathcal{T}_{\mathcal{R}}(X)$ whose objects are equivalence classes of terms $[t] \in T_{\Sigma,E}(X)$ and whose morphisms are equivalence classes of "proof terms" representing proofs in rewriting deduction, i.e., concurrent $\mathcal{R}$-rewrites. The rules for generating such proof terms, with the specification of their respective domain and codomain, are given below; they just "decorate" with proof terms the rules 1-4 of rewriting logic given in Section 3.2. Note that we always use "diagrammatic" notation for morphism composition, i.e., $\alpha;\beta$ always means the composition of α *followed by* β.

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

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

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

$$\frac{\alpha_1 : [t_1] \longrightarrow [t'_1] \quad \dots \quad \alpha_n : [t_n] \longrightarrow [t'_n]}{f(\alpha_1, \dots, \alpha_n) : [f(t_1, \dots, t_n)] \longrightarrow [f(t'_1, \dots, t'_n)]}$$

3. **Replacement**. For each rewrite rule $r : [t(\overline{x}^n)] \longrightarrow [t'(\overline{x}^n)]$ in R,

$$\frac{\alpha_1 : [w_1] \longrightarrow [w'_1] \quad \dots \quad \alpha_n : [w_n] \longrightarrow [w'_n]}{r(\alpha_1, \dots, \alpha_n) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}$$

4. **Composition**.

$$\frac{\alpha : [t_1] \longrightarrow [t_2] \qquad \beta : [t_2] \longrightarrow [t_3]}{\alpha;\beta : [t_1] \longrightarrow [t_3]}$$

Convention and Warning. In the case when the same label r appears in two different rules of R, the "proof terms" $r(\overline{\alpha})$ can sometimes be *ambiguous*. We assume that such ambiguity problems *have been resolved* by disambiguating the label r in the proof terms $r(\overline{\alpha})$ if necessary; with this understanding, we adopt the simpler notation $r(\overline{\alpha})$ to ease the exposition.

Each of the above rules of generation defines a different operation taking certain proof terms as arguments and returning a resulting proof term. In other words, proof terms form an algebraic structure $\mathcal{P}_{\mathcal{R}}(X)$ consisting of a graph with nodes $T_{\Sigma,E}(X)$, with identity arrows, and with operations f (for each $f \in \Sigma$), r (for each rewrite rule), and $\_;\_$ (for composing arrows). Our desired model $\mathcal{T}_{\mathcal{R}}(X)$ is the quotient of $\mathcal{P}_{\mathcal{R}}(X)$ modulo the following equations[25]:

1. **Category**.

   (a) *Associativity.* For all α, β, γ
   $$(\alpha;\beta);\gamma = \alpha;(\beta;\gamma)$$

   (b) *Identities.* For each $\alpha : [t] \longrightarrow [t']$
   $$\alpha;[t'] = \alpha \quad \textit{and} \quad [t];\alpha = \alpha$$

2. **Functoriality of the Σ-algebraic structure**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

   (a) *Preservation of composition.* For all $\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_n$,
   $$f(\alpha_1;\beta_1, \dots, \alpha_n;\beta_n) = f(\alpha_1, \dots, \alpha_n);f(\beta_1, \dots, \beta_n)$$

   (b) *Preservation of identities.*
   $$f([t_1], \dots, [t_n]) = [f(t_1, \dots, t_n)]$$

[25]In the expressions appearing in the equations, when compositions of morphisms are involved, we always implicitly assume that the corresponding domains and codomains match.

3. **Axioms in** E. For $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ an axiom in E, for all $\alpha_1, \ldots, \alpha_n$,

$$t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n)$$

4. **Exchange**. For each $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,

$$\frac{\alpha_1 : [w_1] \longrightarrow [w_1'] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w_n']}{r(\overline{\alpha}) = r(\overline{[w]}); t'(\overline{\alpha}) = t(\overline{\alpha}); r(\overline{[w']})}$$

Note that the set X of variables is actually a parameter of these constructions, and we need not assume X to be fixed and countable. In particular, for $X = \emptyset$, we adopt the notation $\mathcal{T}_{\mathcal{R}}$. The equations in 1 make $\mathcal{T}_{\mathcal{R}}(X)$ a category, the equations in 2 make each $f \in \Sigma$ a functor, and 3 forces the axioms E. The exchange law states that any rewriting of the form $r(\overline{\alpha})$—which represents the *simultaneous* rewriting of the term at the top using rule r *and* "below," i.e., in the subterms matched by the variables, using the rewrites $\overline{\alpha}$—is equivalent to the sequential composition $r(\overline{[w]}); t'(\overline{\alpha})$, corresponding to first rewriting on top with r and then below on the subterms matched by the variables with $\overline{\alpha}$, and is also equivalent to the sequential composition $t(\overline{\alpha}); r(\overline{[w']})$ corresponding to first rewriting below with $\overline{\alpha}$ and then on top with r. Therefore, the exchange law states that rewriting at the top by means of rule r and rewriting "below" using $\overline{\alpha}$ are processes that are independent of each other and can be done either simultaneously or in any order. Since $[t(x_1, \ldots, x_n)]$ and $[t'(x_1, \ldots, x_n)]$ can be regarded as functors $\mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$, from the mathematical point of view the exchange law just asserts that r is a *natural transformation* [62], i.e.,

Lemma 2 For each $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R, the family of morphisms

$$\{r(\overline{[w]}) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w}/\overline{x})] \mid \overline{[w]} \in T_{\Sigma,E}(X)^n\}$$

is a natural transformation $r : [t(x_1, \ldots, x_n)] \Longrightarrow [t'(x_1, \ldots, x_n)]$ between the functors

$$[t(x_1, \ldots, x_n)], [t'(x_1, \ldots, x_n)] : \mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X).$$

□

What the exchange law provides in general is a way of *abstracting* a rewriting computation by considering immaterial the order in which rewrites are performed "above" and "below" in the term; further abstraction among proof terms is obtained from the functoriality equations. The equations 1-4 provide in a sense the *most abstract* "true concurrency" view of the computations of the rewrite theory $\mathcal{R}$ that can reasonably be given. In particular, we can prove that all proof terms have an equivalent expression as a composition of one-step rewrites:

Lemma 3 For each $[\alpha] : [t] \longrightarrow [t']$ in $\mathcal{T}_{\mathcal{R}}(X)$, either $[t] = [t']$ and $[\alpha] = [[t]]$, or there is an $n \in \mathbb{N}$ and a chain of morphisms $[\alpha_i]$, $0 \leq i \leq n$, whose terms α_i describe one-step (concurrent) rewrites

$$[t] \xrightarrow{\alpha_0} [t_1] \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} [t_n] \xrightarrow{\alpha_n} [t']$$

such that $[\alpha] = [\alpha_0; \ldots; \alpha_n]$. In addition, we can always choose all the α_i corresponding to sequential rewrites, i.e., we can decompose $[\alpha]$ into an interleaving sequence. □

The category $\mathcal{T}_{\mathcal{R}}(X)$ is just one among many *models* that can be assigned to the rewrite theory $\mathcal{R}$. The general notion of model, called an $\mathcal{R}$-system, is defined as follows:

Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, an *$\mathcal{R}$-system* $\mathcal{S}$ is a category $\mathcal{S}$ together with:

- a (Σ, E)-algebra structure given by a family of functors

  $$\{f_{\mathcal{S}} : \mathcal{S}^n \longrightarrow \mathcal{S} \mid f \in \Sigma_n, n \in \mathbb{N}\}$$

  satisfying the equations E, i.e., for any $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ in E we have an identity of functors $t_{\mathcal{S}} = t'_{\mathcal{S}}$, where the functor $t_{\mathcal{S}}$ is defined inductively from the functors $f_{\mathcal{S}}$ in the obvious way.

- for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R a natural transformation $r_{\mathcal{S}} : t_{\mathcal{S}} \Longrightarrow t'_{\mathcal{S}}$.

An *$\mathcal{R}$-homomorphism* $F : \mathcal{S} \longrightarrow \mathcal{S}'$ between two $\mathcal{R}$-systems is then a functor $F : \mathcal{S} \longrightarrow \mathcal{S}'$ such that it is a Σ-algebra homomorphism—i.e., $f_{\mathcal{S}} * F = F^n * f_{\mathcal{S}'}$, for each f in Σ_n, $n \in \mathbb{N}$—and such that "F preserves R," i.e., for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R we have the identity of natural transformations[26] $r_{\mathcal{S}} * F = F^n * r_{\mathcal{S}'}$, where n is the number of variables appearing in the rule. This defines a category $\underline{\mathcal{R}\text{-}Sys}$ in the obvious way. □

What the above definition captures formally is the idea that the models of a rewrite theory *are systems*. By a "system" we of course mean a machine-like entity that can be in a variety of *states*, and that can change its state by performing certain *transitions*. Such transitions are of course transitive, and it is natural and convenient to view states as "idle" transitions that do not change the state. In other words, a system can be naturally regarded as a *category*, whose objects are the states of the system and whose morphisms are the system's transitions.

For *sequential* systems such as labelled transition systems this is in a sense the end of the story; such systems exhibit *nondeterminism*, but do not have the required algebraic structure in their states and transitions to exhibit true concurrency [70, 72]. Indeed, what makes a system *concurrent* is precisely the existence of an additional *algebraic structure*. Ugo Montanari and I first observed this fact for the particular case of Petri nets for which the algebraic structure is precisely that of a commutative monoid [81, 80]; this has been illustrated by the `TICKET` example in Section 2.2 where the commutative monoid operation `__` made possible the concurrent firing of several transitions. However, this observation holds in full generality for *any algebraic structure whatever*.

What the algebraic structure captures is twofold. Firstly, *the states themselves are distributed according to such a structure*; for Petri nets the distribution takes the form of a *multiset* that we can visualize with tokens and places; for a functional program involving just syntactic rewriting, the distribution takes the form of a *labelled tree structure* which can be spatially distributed in such a way that many transitions (i.e., rewrites) can happen concurrently in a way analogous to the concurrent firing of transitions in a Petri net. A concurrent object-oriented system as specified by a Maude module combines in a sense aspects of the functional and Petri net examples, because its configuration evolves by multiset *ACI*-rewriting but, underneath such transitions for objects and messages, arbitrarily complex concurrent computations of a functional nature can take place in order to update the values of object attributes as specified by appropriate functional submodules. Secondly, *concurrent transitions are themselves distributed according to the same algebraic structure*; this is what the notion of $\mathcal{R}$-system captures, and

[26]Note that we use diagrammatic order for the *horizontal*, $\alpha * \beta$, and *vertical*, $\gamma ; \delta$, composition of natural transformations (see [62]).

System	$\longleftrightarrow$	*Category*
State	$\longleftrightarrow$	*Object*
Transition	$\longleftrightarrow$	*Morphism*
Procedure	$\longleftrightarrow$	*Natural Transformation*
Distributed Structure	$\longleftrightarrow$	*Algebraic Structure*

Figure 12: The mathematical structure of concurrent systems.

is for example manifested in the concurrent firing of Petri nets, the evolution of concurrent object-oriented systems and, more generally, in any type of concurrent rewriting.

The expressive power of rewrite theories to specify concurrent transition systems[27] is greatly increased by the possibility of having not only transitions, but also *parameterized transitions*, i.e., *procedures*. This is what rewrite rules—with variables—provide. The family of states to which the procedure applies is given by those states where a component of the (distributed) state is a substitution instance of the lefthand side of the rule in question. The rewrite rule is then a *procedure*[28] which transforms the state *locally*, by replacing such a substitution instance by the corresponding substitution instance of the righthand side. The fact that this can take place concurrently with other transitions "below" is precisely what the concept of a *natural transformation* formalizes. The table of Figure 12 summarizes our present discussion.

A detailed proof of the following theorem on the existence of initial and free $\mathcal{R}$-systems for the more general case of conditional rewrite theories is given in [72], where the soundness and completeness of rewriting logic for $\mathcal{R}$-system models is also proved. Below, for $\mathcal{C}$ a category, $\mathit{Obj}(\mathcal{C})$ denotes the set of its objects.

Theorem 4 $\mathcal{T}_{\mathcal{R}}$ is an initial object in the category $\underline{\mathcal{R}\text{-}\mathit{Sys}}$. More generally, $\mathcal{T}_{\mathcal{R}}(X)$ has the following universal property: Given an $\mathcal{R}$-system $\mathcal{S}$, each function $F : X \longrightarrow \mathit{Obj}(\mathcal{S})$ extends uniquely to an $\mathcal{R}$-homomorphism $F^{\natural} : \mathcal{T}_{\mathcal{R}}(X) \longrightarrow \mathcal{S}$. $\Box$

8.2 Preorder, Poset, and Algebra Models

Since $\mathcal{R}$-systems are an "essentially algebraic" concept[29], we can consider classes Θ of $\mathcal{R}$-systems defined by the satisfaction of additional equations. Such classes give rise to full subcategory inclusions $\Theta \hookrightarrow \underline{\mathcal{R}\text{-}\mathit{Sys}}$, and by general universal algebra results about essentially algebraic theories (see, e.g., [14]) such inclusions are *reflective* [62], i.e., for each $\mathcal{R}$-system $\mathcal{S}$ there is an $\mathcal{R}$-system $R_{\Theta}(\mathcal{S}) \in \Theta$ and an $\mathcal{R}$-homomorphism $\rho_{\Theta}(\mathcal{S}) : \mathcal{S} \longrightarrow R_{\Theta}(\mathcal{S})$ such that for any $\mathcal{R}$-homomorphism $F : \mathcal{S} \longrightarrow \mathcal{D}$ with $\mathcal{D} \in \Theta$ there is a unique $\mathcal{R}$-homomorphism $F^{\diamondsuit} : R_{\Theta}(\mathcal{S}) \longrightarrow \mathcal{D}$ such that $F = \rho_{\Theta}(\mathcal{S}); F^{\diamondsuit}$. The assignment $\mathcal{S} \longmapsto R_{\Theta}(\mathcal{S})$ extends to a functor $\underline{\mathcal{R}\text{-}\mathit{Sys}} \longrightarrow \Theta$, called the *reflection functor*.

Therefore, we can consider subcategories of $\underline{\mathcal{R}\text{-}\mathit{Sys}}$ that are defined by certain equations and be guaranteed that they have initial and free objects, that they are closed by subobjects and

[27]Such expressive power is further increased by allowing *conditional* rewrite rules, a more general case to which all that is said in this paper has been extended in [72].

[28]Its *actual parameters* are precisely given by a substitution.

[29]In the precise sense of being specifiable by an "essentially algebraic theory" or a "sketch" [14]; see [72].

products, etc. Consider for example the following equations:

$$\begin{aligned}
&\forall f, g \in \mathit{Arrows},\ f = g \ \ \mathit{if}\ \ \partial_0(f) = \partial_0(g) \wedge \partial_1(f) = \partial_1(g)\\
&\forall f, g \in \mathit{Arrows},\ f = g \ \ \mathit{if}\ \ \partial_0(f) = \partial_1(g) \wedge \partial_1(f) = \partial_0(g)\\
&\forall f \in \mathit{Arrows},\ \partial_0(f) = \partial_1(f).
\end{aligned}$$

where $\partial_0(f)$ and $\partial_1(f)$ denote the source and target of an arrow f respectively. The first equation forces a category to be a preorder, the addition of the second requires this preorder to be a poset, and the three equations together force the poset to be *discrete*, i.e., just a set. By imposing the first one, the first two, or all three, we get full subcategories

$$\underline{\mathcal{R}\text{-}\mathit{Alg}} \subseteq \underline{\mathcal{R}\text{-}\mathit{Pos}} \subseteq \underline{\mathcal{R}\text{-}\mathit{Preord}} \subseteq \underline{\mathcal{R}\text{-}\mathit{Sys}}.$$

A routine inspection of $\underline{\mathcal{R}\text{-}\mathit{Preord}}$ for $\mathcal{R} = (\Sigma, E, L, R)$ reveals that its objects are preordered Σ-algebras $(A, \leq)$ (i.e., preordered sets with a Σ-algebra structure such that all the operations in Σ are monotonic) that satisfy the equations E and such that for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R and for each $\overline{a} \in A^n$ we have, $t_A(\overline{a}) \geq t'_A(\overline{a})$. The poset case is entirely analogous, except that the relation $\leq$ is a partial order instead of being a preorder. Finally, $\underline{\mathcal{R}\text{-}\mathit{Alg}}$ is the category of ordinary Σ-algebras that satisfy the equations $E \cup \mathit{unlabel}(R)$, where the *unlabel* function removes the labels from the rules and turns the sequent signs "$\longrightarrow$" into equality signs.

The reflection functor associated with the inclusion $\underline{\mathcal{R}\text{-}\mathit{Preord}} \subseteq \underline{\mathcal{R}\text{-}\mathit{Sys}}$ sends $\mathcal{T}_{\mathcal{R}}(X)$ to the familiar *$\mathcal{R}$-rewriting relation*[30] $\rightarrow_{\mathcal{R}(X)}$ on E-equivalence classes of terms with variables in X. Similarly, the reflection associated to the inclusion $\underline{\mathcal{R}\text{-}\mathit{Pos}} \subseteq \underline{\mathcal{R}\text{-}\mathit{Sys}}$ maps $\mathcal{T}_{\mathcal{R}}(X)$ to the partial order $\geq_{\mathcal{R}(X)}$ obtained from the preorder $\rightarrow_{\mathcal{R}(X)}$ by identifying any two $[t], [t']$ such that $[t] \rightarrow_{\mathcal{R}(X)} [t']$ and $[t'] \rightarrow_{\mathcal{R}(X)} [t]$. Finally, the reflection functor into $\underline{\mathcal{R}\text{-}\mathit{Alg}}$ maps $\mathcal{T}_{\mathcal{R}}(X)$ to $T_{\mathcal{R}}(X)$, the free Σ-algebra on X satisfying the equations $E \cup \mathit{unlabel}(R)$; therefore, the classical *initial algebra semantics* of (functional) equational specifications reappears here associated with a very special class of models which—when viewed as systems—have only trivial identity transitions.

8.3 The Semantics of Maude

This paper has shown that, by generalizing the logic and the model theory of equational logic to those of rewriting logic, a much broader field of applications for rewrite rule programming is possible—based on the idea of programming *concurrent systems* rather than *algebras*, and including, in particular, concurrent object-oriented programming. The same high standards of mathematical rigor enjoyed by equational logic can be maintained in giving semantics to a language like Maude in the broader context of rewriting logic. We present below a specific proposal for such a semantics having the advantages of keeping functional modules as a sublanguage with a more specialized semantics. Another appealing characteristic of the proposed semantics is that the operational and mathematical semantics of modules are related in a particularly nice way. As already mentioned, all the ideas and results in this paper extend without problem[31] to the *order-sorted* case; the unsorted case has only been used for the sake of a simpler exposition. Therefore, all that is said below is understood in the context of order-sorted rewriting logic.

[30] It is perhaps more suggestive to call $\rightarrow_{\mathcal{R}(X)}$ the *reachability relation* of the system $\mathcal{T}_{\mathcal{R}}(X)$.

[31] Exercising of course the well known precaution of making explicit the universal quantification of rules.

We have already seen that object-oriented modules can be reduced to equivalent system modules having the same behavior but giving a more explicit description of the type structure. Therefore, of the three kinds of modules existing in Maude, namely functional, system and object-oriented, we need only provide a semantics for functional and system modules; they are respectively of the form `fmod` $\mathcal{R}$ `endfm`, and `mod` $\mathcal{R}'$ `endm`, for $\mathcal{R}$ and $\mathcal{R}'$ rewrite theories[32]. Their semantics is given in terms of an *initial machine* linking the module's operational semantics with its denotational semantics. The general notion of a machine is as follows.

For $\mathcal{R}$ a rewrite theory and $\Theta \hookrightarrow \underline{\mathcal{R}\text{-}Sys}$ a reflective full subcategory, an $\mathcal{R}$-*machine over* Θ is an $\mathcal{R}$-homomorphism $[\![\_]\!] : \mathcal{S} \longrightarrow \mathcal{M}$, called the machine's *abstraction map*, with $\mathcal{S}$ an $\mathcal{R}$-system and $\mathcal{M} \in \Theta$. Given $\mathcal{R}$-machines over Θ, $[\![\_]\!] : \mathcal{S} \longrightarrow \mathcal{M}$ and $[\![\_]\!]' : \mathcal{S}' \longrightarrow \mathcal{M}'$, an $\mathcal{R}$-machine *homomorphism* is a pair of $\mathcal{R}$-homomorphisms (F, G), $F : \mathcal{S} \longrightarrow \mathcal{S}'$, $G : \mathcal{M} \longrightarrow \mathcal{M}'$, such that $[\![\_]\!]; G = F; [\![\_]\!]'$. This defines a category $\underline{\mathcal{R}\text{-}Mach/\Theta}$; it is easy to check that the initial object in this category is the unique $\mathcal{R}$-homomorphism $\mathcal{T}_{\mathcal{R}} \longrightarrow R_{\Theta}(\mathcal{T}_{\mathcal{R}})$. $\Box$

The intuitive idea behind a machine $[\![\_]\!] : \mathcal{S} \longrightarrow \mathcal{M}$ is that we can use a *system* $\mathcal{S}$ to *compute* a result relevant for a *model* $\mathcal{M}$ of interest in a class Θ of models. What we do is to perform a certain computation in $\mathcal{S}$, and then output the result by means of the abstraction map $[\![\_]\!]$. A very good example is an *arithmetic machine* with $\mathcal{S} = \mathcal{T}_{\texttt{NAT}}$, for `NAT` the rewriting theory of the Peano natural numbers corresponding to the module `NAT`[33] in Section 2, with $\mathcal{M} = \mathbb{N}$, and with $[\![\_]\!]$ the unique homomorphism from the initial `NAT`-system $\mathcal{T}_{\texttt{NAT}}$; i.e., this is the initial machine in $\underline{\texttt{NAT}\text{-}Mach/\texttt{NAT}\text{-}Alg}$. To compute the result of an arithmetic expression t, we perform a terminating rewriting and output the corresponding number, which is an element of $\mathbb{N}$.

Each choice of a reflective full subcategory Θ as a category of models yields a different semantics. As already implicit in the arithmetic machine example, the *semantics of a functional module*[34] `fmod` $\mathcal{R}$ `endfm` is the initial machine in $\underline{\mathcal{R}\text{-}Mach/\mathcal{R}\text{-}Alg}$. For the *semantics of a system module* `mod` $\mathcal{R}$ `endm` not having any functional submodules, we propose the initial machine in $\underline{\mathcal{R}\text{-}Mach/\mathcal{R}\text{-}Preord}$, but other choices are also possible. On the one hand, we could choose to be as concrete as possible and take $\Theta = \underline{\mathcal{R}\text{-}Sys}$ in which case the abstraction map is the identity homomorphism for $\mathcal{T}_{\mathcal{R}}$. On the other hand, we could instead be even more abstract, and choose $\Theta = \underline{\mathcal{R}\text{-}Pos}$; however, this would have the unfortunate effect of collapsing all the states of a cyclic rewriting, which seems inappropriate for many "reactive" systems. If the machine $\mathcal{T}_{\mathcal{R}} \longrightarrow \mathcal{M}$ is the semantics of a functional or system module with rewrite theory $\mathcal{R}$, then we call $\mathcal{T}_{\mathcal{R}}$ the module's *operational semantics*, and $\mathcal{M}$ its *denotational semantics*. Therefore, the operational and denotational semantics of a module can be extracted from its initial machine semantics by projecting to the domain or codomain of the abstraction map. This makes Maude a *logic programming language* in the general axiomatic sense of [68] mentioned in Section 7.

9 Related Work

Within the limits of this paper it is impossible to do justice to the wealth of related literature on concurrent object-oriented programming, term and graph rewriting, abstract data

[32]Note that, although in a functional module `fmod` $\mathcal{R}$ `endfm`, $\mathcal{R}$ is an *equational* theory, we can regard $\mathcal{R}$ as a rewrite theory whose rules are its Church-Rosser equations, and whose structural axioms E are those axioms modulo which we are rewriting, such as associativity, commutativity, identity .... Note also that in the case of system modules having functional submodules, which is treated in [70], the semantics given below would be inaccurate, because we must "remember" that the submodule in question is functional.

[33]In this case E is the commutativity attribute, and R consists of the two rules for addition.

[34]For this semantics to behave well, the rules R in the functional module $\mathcal{R}$ should be *confluent* modulo E.

types, concurrency theory, Petri nets, linear and equational logic, ordered, continuous and nondeterministic algebras, etc. The paper [72] contains 125 such references. In the area of concurrent object-oriented programming alone there are many references; one may start with [105, 4, 103, 101] and references there, plus papers in recent OOPSLA and ECOOP proceedings, and of course this volume. Here the attempt will be much more restricted, and will only try to briefly discuss a limited amount of work in some related areas.

FOOPS. Since the work on FOOPS and FOOPLog [49] was the first attempt, in joint work with Joseph Goguen, to achieve the two ends of providing a semantic framework for object-oriented programming and of unifying functional, relational, and object-oriented programming, and since in fact the ideas and experience of FOOPS have had an important influence on Maude, a few remarks should be made by way of comparison between the two languages.

Although both projects share the two ends just mentioned, their semantic frameworks—which provide the means to attain those ends—are quite different. There are indeed important similarities, such as the use of order-sorted techniques, the sharing of OBJ as a functional sublanguage, the basic agreement on modularity and parameterization techniques, and the shared distinction between the levels of sorts, classes and modules. However, the semantic frameworks actually developed are so different that a comparison amounting to something like an explanation of one framework in terms of the other seems quite difficult.

The original semantic framework of FOOPS given in [49] used two different semantic accounts. One based on order-sorted equational logic with hidden sorts—using the ideas of [47] and [77] about algebraic data types with state which have been further developed in [42, 54]—and another based on reflective equational logic. The two accounts complemented each other well, but neither of them was comprehensive enough to warrant abandoning the other. For example, creation and deletion of objects and interactions between objects are not covered by the hidden sort account but have a good explanation in terms of reflective equational logic. Recent work by Joseph Goguen and his coworkers at Oxford has developed a third semantic account of FOOPS in terms of sheaves [43, 55] which is very useful for treating concurrency aspects not covered in [49].

In spite of the many good contributions provided by FOOPS, the reason that motivated the present work was a vague dissatisfaction with the somewhat limited way in which the two goals of providing a semantic framework for object-oriented programming and of unifying functional, relational, and object-oriented programming were achieved by the FOOPS framework. On the one hand, the need to rely on several semantic accounts suggested seeking a simple logical framework that could fully account for all aspects of object-oriented systems, including their concurrency; on the other, having a logic on which concurrent object-oriented programming could be defined as logic programming in the strict axiomatic sense of [68] and on which the functional and relational paradigms could be unified by maps of logics seemed the most satisfactory way of achieving the desired unification of paradigms. Although FOOPS and FOOPLog are certainly declarative languages, for the moment they fall short of being logic programming languages in the precise axiomatic sense of [68], although this does not exclude that they could be shown to be so in the future.

Algebraic Approaches to Object-Oriented Programming. Besides the recent work on FOOPS at Oxford already discussed, there is also important work on the semantics of object-oriented programming using algebraic techniques by members of the ESPRIT research group

IS-CORE including Amílcar Sernadas, Hans-Dieter Ehrich, Udo Lipeck, Tom Maibaum, Robert Meersman, and their collaborators (see [95] for a collection of papers), and there is joint work by Hans-Dieter Ehrich, Joseph Goguen and Amílcar Sernadas relating Goguen's sheaf semantics to ideas in the IS-CORE group [34]. Although we can generally say that this body of work makes use of algebraic data type techniques and aims at a conceptual clarification of the notion of object, a detailed comparison of the work of the IS-CORE group with the present work is beyond the scope of this paper.

There is also important work by Milner, Parrow and Walker on the π-calculus [86] giving a semantics to actors in a generalization of Milner's CCS. As the recent work of Milner shows [85], the π-calculus, or at least a good part of it, seems to be naturally expressible in terms of Berry and Boudol's Chemical Abstract Machine [15] and therefore by rewrite rules modulo *ACI*. This suggests that the π-calculus, or important fragments of it, can be viewed naturally as a specialization of the concurrency model provided by rewriting logic; however, a more detailed comparison will have to wait for a future occasion.

More generally, there is a broad body of work using a variety of algebraic approaches in the specification of concurrency that has been recently surveyed thanks to the efforts of Egidio Astesiano and Gianna Reggio [10].

Type Theory Approaches. In recent years there has been a steady increase in the use of constructive type theory to give semantics to some aspects of object-oriented programming. Early important references include the original work of Reynolds on subtypes [94], Cardelli's paper on the semantics of inheritance [21], and the paper of Cardelli and Wegner on the FUN language [24]; a good number of other papers by a variety of authors have been written since then. For the most part, work in this area seems to take what might be described as a "translational" approach in which, by using increasingly more powerful constructive type theories, aspects and features of object-oriented programming are translated into type-theoretic accounts of them. In this way, a number of aspects have been studied, such as, for example, applicative models of inheritance (e.g., [22, 19, 17, 91]), record operations (e.g., [23]), reuse aspects of inheritance (e.g., [29, 88, 18]), and type-checking issues (e.g., [87, 31, 61]).

Although valuable contributions clarifying some aspects of object-oriented programming have been made by proponents of this approach, it seems for the moment uncertain whether the general approach of translating object-oriented programming features into type-theoretic formulations will ultimately succeed in covering in a satisfactory way all the aspects of object-oriented programming. Judging from the experience of previous work on the denotational semantics of imperative languages—of which these type-theoretic approaches are in a sense a further development—one might be inclined to think that the addition of concurrency, on which these approaches seem to be silent, could prove quite hard. Since most constructive type theories have their origins in the foundations of mathematics and were designed to deal with unchanging mathematical entities[35], the difficulties in dealing with action and change in most of them can be quite intrinsic. In addition, a difficulty with some of the current work in this area is the sheer complexity of the formalisms into which object-oriented concepts are sometimes translated, which makes at times difficult to understand whether a given account does justice to and sheds light on the original problem, and raises some doubts about whether a simpler account should instead be sought. One of the proponents of this approach seems to acknowledge to some extent this difficulty in the statement [18],

[35]One notable exception is Girard's Linear Logic [39].

"While the semantics of our language is rather complex, involving fixed points at both the element and the type level, we believe that this complexity underlies the basic concepts of object-oriented programming languages."

Maude—and for that matter FOOPS [49] and OBJ [53]—shares with type-theoretic approaches the use of *subtypes*. In those three languages the typing is order-sorted and uses ideas that go back to the original paper by Goguen [41] and have been further developed by several authors (see [51] and references there), whereas the theories of subtypes used in type-theoretic approaches can be traced back to the work of Reynolds on implicit conversions and generic operators [94]. Although there are some differences between both approaches—for example, the notion of subtype as inclusion tends to be lost in type-theoretic approaches—the two are in fact closely related, as recent joint work with Narciso Martí-Oliet [64] has demonstrated by extending the logic and the model theory of order-sorted algebra to higher types in an approach that subsumes those of [51] and [94] while maintaining a distinction between inclusive and coercive notions of subtype. However, by making the option of having higher-order types orthogonal to subtyping issues, the order-sorted approach is in fact a considerably simpler formalism and affords a simpler treatment of object-oriented concepts.

In the case of Maude, one essential difference is that order-sortedness pervades not only equational logic, which is used for functional modules, but also rewriting logic which is the logic used to give semantics to system modules and to object-oriented modules; therefore, concurrent actions—which would be problematic in a functional context—mesh very well with subclasses in a rewriting logic account that, as shown in this paper, provides a simple semantic framework for object-oriented programming.

Logic Programming Approaches. The theoretical work ahead should include an exploration of how the approach presented here for MaudeLog can be precisely related to other recent (relational) logic programming approaches such as *concurrent logic programming* (see the survey [97]), which is in a sense related to object-oriented programming [98], the work of Corradini and Montanari [30] which addresses concurrency issues and also belongs to this general area, the work of Saraswat and others on *concurrent constraint programming* [96], and the work of Andreoli and Pareschi on *linear logic programming* [8, 9], which is directly aimed at supporting concurrent object-oriented computations. Since quantifier-free linear logic can be regarded as a special case of rewriting logic [63], this last approach, though different, is somewhat closer in spirit to Maude and MaudeLog. Detailed comparisons of this kind may also provide additional semantic insights on issues hitherto addressed in a more operational or proof-theoretic way in some of these other approaches.

10 Concluding Remarks

This paper has presented a general semantic framework for object-oriented programming based on rewriting logic that is intrinsically concurrent and provides a simple and rigorous account of key concepts such as objects, classes and class inheritance, concurrent object-oriented synchronous and asynchronous communication, autonomous objects, and object creation and deletion. A declarative version of the actor model has been obtained as a special case of the framework. In addition, the Maude language—based on rewriting logic—which unifies in a fully declarative way the functional and concurrent object-oriented paradigms has been introduced, and its features and expressive power have been illustrated with examples. It has been explained that in Maude the simultaneous support of concurrent communication and class inheritance is entirely unproblematic. The parameterization and modularity mechanisms of Maude have been illustrated with examples and it has been shown how such mechanisms can support flexible ways of code reuse that complement those of class inheritance. The Simple Maude sublanguage has been described, and its capabilities as a machine independent parallel language with support for multilingual extensions and open heterogeneous systems, as well as implementation ideas for mapping it on several parallel architectures have been discussed. The logic and model theory of rewriting logic have been presented and have been used to give a mathematical semantics to Maude modules. Finally, an extension of Maude called MaudeLog that is also based on rewriting logic has been proposed as a fully declarative unification of the functional, relational, and concurrent object-oriented paradigms.

The present paper is a report of work in progress. Although the logical foundations of Maude are well established, much remains to be done to move the ideas forward in several directions, including the following:

- More experience should be gained with examples to advance the language design and to further explore the capabilities of the language. This will also help in sharpening and expanding the boundaries of Maude and Simple Maude.

- Applications of Maude and MaudeLog and of the rewriting logic formalism to a number of areas such as distributed artificial intelligence[36], object-oriented databases[37], communication protocols, and discrete event simulation should be investigated and illustrated with examples.

- Specification and verification aspects should be studied and should be illustrated with examples. The specification logics should not be limited to equational and rewriting logic. Rewriting logic should be embedded within a richer logic to be used for specification purposes. This will increase the expressiveness of Maude's functional, system and object-oriented theories.

- The study of parameterization and modularity issues, already initiated in [71] and [70] respectively, and of the module inheritance techniques discussed in this paper should be advanced.

- The work on program transformations to derive more efficient and more easily implementable modules from less efficient ones or even from specifications should be continued and advanced. More generally, the design of machine-independent program analysis and program optimization tools based on the concurrent rewriting model should be explored.

- Implementation and compilation techniques for various classes of parallel architectures should be studied in more detail, trying to achieve the greatest possible degree of portability and genericity across different machine implementations. A Maude interpreter should be developed, as well as a portable parallel implementation of Simple Maude.

- Multilingual extensions and uses of Maude in the context of open heterogeneous systems should be studied in greater detail.

[36]For a Maude example of distributed coordination between agents see Section 7 of [75].

[37]A first step in applying rewriting logic to the semantics of object-oriented databases has been taken in [82].

- All aspects of MaudeLog should be further developed; much work remains to be done on its language design, on its underlying theory and relations to other approaches, on its deduction and operational semantics, and eventually on an actual implementation.

Acknowledgements

It is a pleasure to thank my fellow members of the Declarative Languages and Architecture Group at SRI International, especially Timothy Winkler, Narciso Martí-Oliet and Patrick Lincoln. The present exposition has benefited very much from many discussions with them, from their comments, and from their technical contributions to Maude. In addition, Timothy Winkler has kindly allowed me to summarize our joint work on Maude [83] and on broadcasting communication and has suggested many improvements; Narciso Martí-Oliet has caught many small errors, has suggested many improvements to the text, and has influenced the rewriting logic ideas through our joint work on linear logic and its relationship to Petri nets [65], and on action and change in rewriting logic [63]; and Patrick Lincoln's contributions to program transformations for Maude's object-oriented modules as well as his comments and suggestions have also influenced the paper.

It is also a pleasure to thank Joseph Goguen for our long term collaboration on the OBJ, Eqlog and FOOPS languages [45, 53, 49], concurrent rewriting [46] and its implementation on the RRM architecture [50, 6], all of which have directly influenced this work, and Prof. Ugo Montanari for our joint work on the semantics of Petri nets [80, 81] that was an important early influence on rewriting logic.

I cordially thank Akinori Yonezawa and Satoshi Matsuoka for kindly explaining to me the difficulties involved in the "inheritance anomaly" along several conversations, and Peter Wegner, Gul Agha, and Kristen Nygaard for their kindness and interest in listening to my ideas about object-oriented programming and for their helpful comments and suggestions.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. In *Proc. POPL'90*, pages 31–46. ACM, 1990.

[2] G. Agha. *Actors.* MIT Press, 1986.

[3] 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.

[4] G. Agha, P. Wegner, and A. Yonezawa, editors. *Proceedings of the ACM-SIGPLAN Workshop on Object-Based Concurrent Programming.* ACM Sigplan Notices, April 1989.

[5] 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.

[6] 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.

[7] 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.

[8] Jean-Marc Andreoli and Remo Pareschi. LO and behold! Concurrent structured processes. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 44–56. ACM, 1990.

[9] Jean-Marc Andreoli and Remo Pareschi. Communication as fair distribution of knowledge. In *OOPSLA'91 Conference on Object-Oriented Programming, Phoenix, Arizona, October 1991*, pages 212–229. ACM, 1991.

[10] Egidio Astesiano and Gianna Reggio. Algebraic specification of concurrency. To appear in *Proceedings of the ADT'91 Workshop*, Springer LNCS, 1992.

[11] W. Athas and C. Seitz. Multicomputers: Message-passing concurrent computers. *Computer*, pages 9–24, August 1988.

[12] J.-P. Banâtre, A. Coutant, and D. Le Mètayer. Parallel machines for multiset transformation and their programming style. *Informationstechnik* **it**, 30(2):99–109, 1988.

[13] J.-P. Banâtre and D. Le Mètayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[14] M. Barr and C. Wells. *Toposes, Triples and Theories.* Springer-Verlag, 1985.

[15] Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. In *Proc. POPL'90*, pages 81–94. ACM, 1990.

[16] Graham Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin.* Charwell-Bratt Ltd, 1979.

[17] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Proc. LICS'89*, pages 112–129. IEEE, 1989.

[18] Kim Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. Technical Report CS-92-01, Williams College, January 1992.

[19] Kim Bruce and Giuseppe Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87:196–240, 1990.

[20] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer LNCS 86, 1980.

[21] Luca Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types, LNCS 173*, pages 51–67. Springer LNCS 173, 1984.

[22] Luca Cardelli. Structural Subtyping and the Notion of Power Type. In *Proc. POPL'88*. ACM, 1988.

[23] Luca Cardelli and John Mitchell. Operations on records. *Math. Struct. in Comp. Sci.*, 1:3–48, 1991.

[24] Luca Cardelli and Peter Wegner. On understanding types, data abstracton and polymorphism. *Computing Surveys*, 17:471–522, 1985.

[25] N. Carriero and D. Gelernter. Linda in context. *Communications of the Association for Computing Machinery*, 32:444–458, April 1989.

[26] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[27] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, 1992.

[28] Will Clinger. Foundations of actor semantics. Technical report AI-TR-633, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1981.

[29] William Cook, Walter Hill, and Peter Canning. Inheritance is not Subtyping. In *Proc. POPL'90*, pages 125–135. ACM, 1990.

[30] Andrea Corradini and Ugo Montanari. An algebraic semantics of logic programs as structured transition systems. In S. Debray and M. Hermenegildo, editors, *North American Conference on Logic Programming*, pages 788–812. MIT Press, 1990.

[31] Pierre-Louis Curien and Giorgio Ghelli. Coherence and subsumption. To appear in *Mathematical Structures in Computer Science*, 1991.

[32] Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. The simula 67 common base language. Technical report, Norwegian Computing Center, Oslo, 1970. Publication S-22.

[33] 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.

[34] Hans-Dieter Ehrich, Joseph Goguen, and Amílcar Sernadas. A categorical theory of objects a observed processes. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, Noordwijkerhout, The Netherlands, May/June 1990*, pages 203–228. Springer LNCS 489, 1991.

[35] J. Engelfriet, G. Leih, and G. Rozenberg. Parallel object-based systems and Petri nets, I and II. Technical Report 90-04,90-05, Dept. of Computer Science, University of Leiden, February 1990.

[36] J. Engelfriet, G. Leih, and G. Rozenberg. Net-based description of parallel object-based systems, or POTs and POPs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, Noordwijkerhout, The Netherlands, May/June 1990*, pages 229–273. Springer LNCS 489, 1991.

[37] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[38] 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.

[39] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[40] Jean-Yves Girard. Towards a geometry of interaction. In J.W. Gray and A. Scedrov, editors, *Proc. AMS Summer Research Conference on Categories in Computer Science and Logic, Boulder, Colorado, June 1987*, pages 69–108. American Mathematical Society, 1989.

[41] Joseph Goguen. Order sorted algebra. Technical Report Semantics and Theory of Computation Report 14, UCLA, 1978.

[42] Joseph Goguen. Types as theories. In G.M. Reed, A.W. Roscoe, and R. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford University Press, 1991.

[43] Joseph Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, 2(2):159–191, 1992.

[44] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[45] Joseph Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy Winkler. An introduction to OBJ3. In Jean-Pierre Jouannaud and Stephane Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting, Orsay, France, July 8-10, 1987*, pages 258–263. Springer LNCS 308, 1988.

[46] 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.

[47] Joseph Goguen and José Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E. M. Schmidt, editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer LNCS 140, 1982.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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, 1993. To appear in J.A. Goguen, editor, *Applications of Algebraic Specification Using OBJ*, Cambridge University Press.

[54] 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.*

[55] Joseph Goguen and David Wolfram. A sheaf semantics for FOOPS expressions (extended abstract). In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing*, pages 81–98. Springer LNCS 612, 1992.

[56] Gerard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27:797–821, 1980. Preliminary version in *18th Symposium on Mathematical Foundations of Computer Science*, 1977.

[57] R. Jagannathan and A.A. Faustini. The GLU programming language. Technical Report SRI-CSL-90-11, SRI International, Computer Science Laboratory, November 1990.

[58] Gregor Kiczales, Jim des Riviers, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[59] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In T. Lepistö and A. Salomaa, editors, *Proceedings, 15th Intl. Coll. on Automata, Languages and Programming, Tampere, Finland, July 11-15, 1988*, pages 287–301. Springer LNCS 317, 1988.

[60] Simon S. Lam and A. Udaya Shankar. A relational notation for state transition systems. *IEEE Transactions on Software Engineering*, SE-16(7):755–775, July 1990.

[61] Patrick Lincoln and John Mitchell. Algorithmic aspects of type inference with subtypes. In *Proc. POPL'92*. ACM, 1992.

[62] Saunders MacLane. *Categories for the working mathematician.* Springer-Verlag, 1971.

[63] Narciso Martí-Oliet and José Meseguer. Action and change in rewriting logic. Paper in preparation. Given as a talk at the ESPRIT Workshop on Logics of Action and Change, Lisbon, January 1993.

[64] Narciso Martí-Oliet and José Meseguer. Inclusions and subtypes. Technical Report SRI-CSL-90-16, SRI International, Computer Science Laboratory, December 1990. Submitted for publication.

[65] Narciso Martí-Oliet and José Meseguer. From Petri nets to linear logic through categories: a survey. *Intl. J. of Foundations of Comp. Sci.*, 2(4):297–399, 1991.

[66] Satoshi Matsuoka, Takuo Watanabe, Yuuji Ichisugi, and Akinori Yonezawa. Object-oriented concurrent reflective architectures. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing*, pages 211–226. Springer LNCS 612, 1992.

[67] 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.

[68] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[69] 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.

[70] José Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of the Concur'90 Conference, Amsterdam, August 1990*, pages 384–400. Springer LNCS 458, 1990.

[71] José Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, February 1990. Revised June 1990.

[72] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. Also Technical Report SRI-CSL-91-05, SRI International, Computer Science Laboratory, February 1991.

[73] José Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Proc. 3rd Intl. Conf. on Algebraic and Logic Programming*, pages 158–200. Springer LNCS 632, 1992.

[74] José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. Technical Report SRI-CSL-92-14, SRI International, Computer Science Laboratory, 1992. To appear in *Proc. ECOOP'93*, Springer LNCS.

[75] José Meseguer, Kokichi Futatsugi, and Timothy Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. In *Proceedings of the 1992 International Symposium on New Models for Software Architecture, Tokyo, Japan, November 1992*, pages 61–106. Research Institute of Software Engineering, 1992.

[76] José Meseguer and Joseph Goguen. Order-sorted algebra II. In preparation.

[77] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.

[78] José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. Technical Report SRI-CSL-90-06, SRI International, Computer Science Laboratory, June 1990. To appear in *Information and Computation.*

[79] José Meseguer, Joseph Goguen, and Gert Smolka. Order-sorted unification. *J. Symbolic Computation*, 8:383–413, 1989.

[80] 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.

[81] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990. Appeared as Technical Report SRI-CSL-88-3, SRI International, Computer Science Laboratory, January 1988.

[82] José Meseguer and Xiaolei Qian. A logical semantics for object-oriented databases. Technical Report SRI-CSL-92-15, SRI International, Computer Science Laboratory, 1992. To appear in *Proc. International SIGMOD Conference on Management of Data*, May 1993, ACM.

[83] 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. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.

[84] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[85] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[86] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes I and II. Technical Report ECS-LFCS-89-85&86, Dept. of Computer Science, University of Edinburgh, 1989.

[87] John Mitchell. Coercion and type inference. In *Proc. POPL'84*, pages 175–185. ACM, 1984.

[88] John Mitchell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proc. POPL'90*, pages 109–124. ACM, 1990.

[89] Kristen Nygaard. Basic concepts in object-oriented programming. Lecture and paper delivered at the Object-Oriented Programming Workshop held at Yorktwon Heights, New Yor, June 9-13, 1986; abstract in *Sigplan Notices*, 21, No. 10, page 187, October 1986, 1986.

[90] M. Papathomas. Concurrency issues in object-oriented programming languages. In D. Tsichritzis, editor, *Object Oriented Development*, pages 207–246. Université de Geneve, 1989.

[91] Wesley Phoa. Using fibrations to understand subtypes. To appear in M. Fourman, P. Johnstone, and A. Pitts (eds.) *Proc. Symp. on Applications of Categories in Computer Science, Durham, 1991*, Cambridge University Press, 1992.

[92] Sara Porat and Nissim Francez. Fairness in term rewriting systems. Manuscript, Technion, May 3, 1990.

[93] Wolfgang Reisig. *Petri Nets.* Springer-Verlag, 1985.

[94] John Reynolds. Using category theory to design implicit conversions and generic operators. In Neal D. Jones, editor, *Semantics Directed Compiler Generation*, pages 211–258. Springer LNCS 94, 1980.

[95] G. Saake and A. Sernadas, editors. *Information Systems—Correctness and Reusability.* Technische Universität Braunschweig, Information-Berichte 91-03, 1991.

[96] Vijay Saraswat. *Concurrent constraint programming languages.* PhD thesis, Computer Science Department, Carnegie-Mellon University, 1989.

[97] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21:413–510, 1989.

[98] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in concurrent Prolog. *New Generation Computing*, 1:25–48, 1983.

[99] Brian Smith and Akinori Yonezawa, editors. *Proc. of the IMSA'92 International Workshop on Reflection and Meta-Level Architecture, Tokyo, November 1992.* Research Institute of Software Engineering, 1992.

[100] Sophie Tison. Fair termination is decidable for ground systems. In Nachum Dershowitz, editor, *Rewriting Techniques and Applications, Chappel Hill, North Carolina*, pages 462–476. Springer LNCS 355, 1989.

[101] M. Tokoro, O. Nierstrasz, and P. Wegner, editors. *Object-Based Concurrent Computing.* Springer LNCS 612, 1992.

[102] S. Watari, S. Kono, E. Osawa, R. Smoody, and M. Tokoro. Extending object-oriented systems to support dialectic worldviews. In *Symposium on Advanced Database Systems, Kyoto, Japan, December 1989*, 1989.

[103] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1:7–87, 1990.

[104] 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.

[105] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, 1988.