A Logical Theory of Concurrent Objects*

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 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. Thanks to this axiomatization, it becomes possible to study the behavior of concurrent objects by formal methods in a logic intrinsic to their computation. The relationship with Actors and with other models of concurrent computation is also discussed. A direct fruit of this theory is a new language, called Maude, to program concurrent object-oriented modules in an entirely declarative way using rewriting logic; modules written in this language are used to illustrate the main ideas with examples. Maude contains OBJ3 as a functional sublanguage and provides a simple and semantically rigorous integration of functional programming and concurrent object-oriented programming.

1 Introduction

The strong historical influence of mathematics on logic during the 19th and 20th centuries, while providing logic with high standards of rigor, has had the limiting effect of developing logic in a timeless, Platonic, direction that is not well suited for the dynamical nature of computation. For logic programming languages this is felt as an inadequacy to deal, within pure logic, with dynamic aspects of computation such as input-output, concurrency, or asserting new database facts. This applies to functional languages —based on a first order or higher order version of equational logic— and also to relational languages such as Prolog. This state of affairs poses an unhealthy *dualistic dilemma*, of a gnostic kind, forcing one to choose between a clean, timeless, world of logic and the dirty material world of change and chaos.

This paper proposes a concrete solution to such a dilemma in an area where the dynamic aspects of computation are paramount, namely concurrent object-oriented programming. The paper presents a logic, called *rewriting logic*, that is ideally suited for concurrent object-oriented computation, in the sense that computation of this kind can be identified with deduction in this logic. It also presents a programming language, called Maude, that is directly based on rewriting logic in the sense that a Maude program module is a theory in that logic. Maude contains OBJ3 [12] as a functional sublanguage and supports an entirely declarative style of programming concurrent object-oriented systems. This is illustrated through a variety of examples presented in the paper.

One of the basic inadequacies of traditional logics for dealing with change is the so called *frame problem*, which forces such logics to make explicit many contextual aspects not affected by a change. Because of the local character and flexibility of its logical axioms, rewriting logic deals with context in an implicit way. Such flexibility and freedom from context is even greater in the case of axioms specifying object-oriented systems; this is because in that case the context takes the form of a highly unstructured distributed configuration in which objects and messages "float." Technically, this flexible ensemble of objects and messages is held together by an associative and commutative union operator; communication events occur concurrently in this configuration, with each such event being a context independent localized rewriting of objects and messages by means of an axiom. This makes rewriting logic very well suited to program *open systems* —which can be embedded in a rich and open-ended variety of contexts— in a declarative way.

The paper motivates the basic ideas of concurrent rewriting with Maude examples, presents the rules of rewriting logic, develops a logical theory of concurrent objects based on that logic, discusses Maude's object-oriented modules and a number of issues in concurrent object-oriented programming, including Actors, and gives a model-theoretic semantics for Maude based on a model theory for rewriting logic. The final section discusses related work and summarizes the main points of the paper.

Acknowledgements. I specially thank Prof. Joseph Goguen for our long term collaboration on the OBJ and FOOPS languages [12, 14], concurrent rewriting [13] and its implementation on the RRM architecture [15, 17], all of which have directly influenced this work. I specially thank Prof. Ugo Montanari for our collaboration on the semantics of Petri nets [27, 28] that has been a source of inspiration for the more general ideas presented here. Mr. Narciso Martí-Oliet deserves special thanks for our collaboration on the semantics of linear logic and its relationship to Petri nets [23, 22], which is another source of inspiration for this work; he also provided very many helpful comments and suggestions for improving the exposition. I also thank all my fellow members of the OBJ and RRM teams, past and present, and in particular Mr. Timothy Winkler, who deserves special thanks for his many very good comments about the technical content as well as for his kind assistance with the pictures.

2 Concurrent Rewriting

Concurrent rewriting is motivated with examples of *functional* and *system* modules in Maude. The system module

*Supported by Office of Naval Research Contracts N00014-90-C-0086 and N00014-88-C-0618, and NSF Grant CCR-8707155.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 0-89791-411-2/90/0010-0101...$1.50

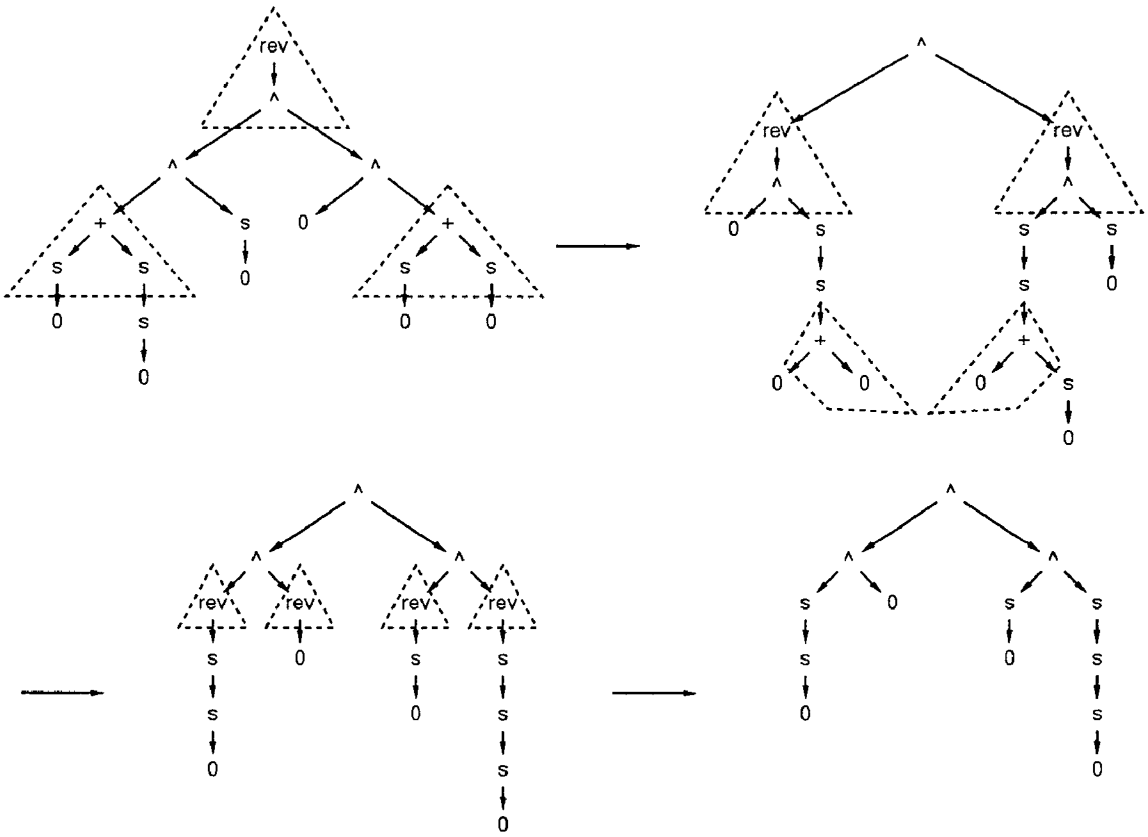


Figure 1: Concurrent rewriting of a tree of numbers.

examples show that the traditional interpretation of rewrite rules as equations must be abandoned.

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 [12]:

```
fmod NAT is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
endfm
```

```
fmod NAT-REVERSE is
  protecting NAT .
  sort Tree .
  subsorts Nat < Tree .
  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, 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 and subsorts stating the different sorts of data manipulated by the module and how those sorts are related. As OBJ3's, Maude's type structure is *order-sorted* [16]; therefore, it is possible to declare one sort as a *subsort* of another; for example, 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 function s_), "infix" (_+_) or any "mixfix" combination as well as standard parenthesized notation (rev). Variables to be used for defining equations are declared with their corresponding types, and then equations are given; such equations provide the actual "code" of the module. 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[1] [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, which avoids having to program it explicitly. Since in the two modules above the equations are confluent and terminating (see [20, 7] for a definition of these notions) the *order* in which the rules are applied does not affect at all the final result.

As in OBJ3, functional modules can be *parameterized*. For example, we can define a parameterized module for lists whose elements belong to a parameter set of elements. In such modules, the properties that the parameter must satisfy are specified by one or more (functional) *parameter theories*. In this case, the 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

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

Note that the "empty syntax" operator __ has been declared *associative* and has the constant nil as its *identity* element. Rewriting with this module is performed *modulo* associativity and identity; this means that we can disregard parentheses and that nil can match a List variable. For example, if we instantiate this module to form lists of naturals by means of the module expression

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

then the second equation for length will match the expression length(7) modulo associativity and identity by matching E to 7 and L to nil. Another example of a parameterized module can be obtained 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.

As in OBJ3, the denotational semantics of functional modules is given by the *initial algebra*[2] associated to the syntax and equations in the module [18, 16], i.e., associated to the *equational theory* that the module represents. Up to now, most work on term rewriting has dealt with that case. However, the true possibilities of the concurrent rewriting model are by no means restricted to this case; we consider below a very important class of Maude modules, called *system modules*, that cannot be dealt with within the initial algebra framework.

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 NAT-CHOICE module below, which adds a nondeterministic choice operator to the natural numbers.

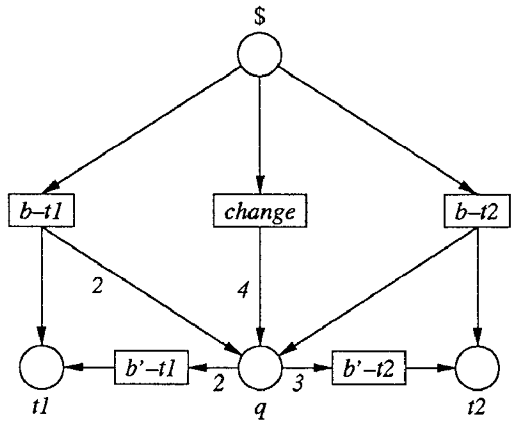
```
mod NAT-CHOICE is
  extending NAT .
  op _?_ : Nat Nat -> Nat .
  vars N M : Nat .
  rl N ? M => N .
  rl N ? M => M .
endm
```

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, 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 initial algebra semantics. My proposal is to seek 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

[1] 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.

[2] For example, the initial algebra of the NAT module is of course the natural numbers with successor and addition.



```
mod TICKET is
  sort Place .
  ops $,q,t1,t2 : -> Place .
  op _⊗_ : Place Place -> Place
            [assoc comm id: λ] .
  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 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 λ. Therefore, concurrent rewriting in this module is performed *modulo* the associativity, commutativity and identity axioms for ⊗. We call such a rewriting *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:

$$\begin{aligned} &\mathtt{q{\otimes}q{\otimes}q{\otimes}q{\otimes}\$\otimes\$} \longrightarrow \mathtt{q{\otimes}q{\otimes}t1{\otimes}t2{\otimes}q{\otimes}\$} \\ &\quad\longrightarrow \mathtt{t2{\otimes}t1{\otimes}t2{\otimes}t2{\otimes}q.} \end{aligned}$$

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

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, I 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 $n \in \mathbb{N}$. I 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. We let $t =_E t'$ denote the congruence modulo E of two terms t, t', and $[t]_E$ or just $[t]$ denote the E-equivalence class of t.

Given a term $t \in T_\Sigma(\{x_1, \ldots, x_n\})$, and a sequence of 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, I 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, $t(u_1/x_1, \ldots, u_n/x_n)$ is abbreviated to $t(\overline{u}/\overline{x})$.

3.2 Rewriting Logic

We are now ready to introduce the new logic that we are seeking, which I 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 a given 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., [20, 7].)

Given a signature (Σ, E), the *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.

Definition 1 A *(labelled) rewrite theory*[3] $\mathcal{R}$ 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 called the 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*[4]. 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[5] $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, 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 $[t] \in T_{\Sigma,E}(X)$,

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

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

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

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

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

4. **Transitivity**.

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

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*.

In rewriting logic a sequent $[t] \longrightarrow [t']$ should not be read as "$[t]$ *equals* $[t']$," but as "$[t]$ *becomes* $[t']$." Therefore, rewriting logic is a logic of *becoming* or *change*, not a logic of equality in a static Platonic sense. Adding the symmetry rule is 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 *logical connectives* in Σ— that asserts being in a certain *state* having a certain *structure*. The rules of rewriting logic are therefore rules to reason about *change in a concurrent system*. 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.

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.

Definition 2 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 $[t] \longrightarrow [t']$. The notions of sequential and deterministic rewrite theory can be made relative to a given subset $S \subseteq T_{\Sigma,E}(X)$ by requiring that the corresponding property holds for each $[t']$ "reachable from S," i.e., for each $[t']$ such that for some $[t] \in S$ there is a concurrent $\mathcal{R}$-rewrite $[t] \longrightarrow [t']$. □

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 [25]. Indeed, concurrent rewriting is a more convenient way of considering such notions than the traditional way using sequential rewriting.

4 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 our 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.

[3] I consciously depart from the standard terminology, that would call $\mathcal{R}$ a *rewrite system*. The reason for this departure is very specific. I want to keep the term "rewrite system" for the *models* of such a theory, which will be defined in Section 5 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 formalize our dynamic intuitions about rewriting.

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

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

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

[5] 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.

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* is represented as a term

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

where O is the object's name or identifier, C is its class, the a_i's are the names of the object's *attributes*, and the v_i's are their corresponding *values*; in particular, object *identifiers* can be values. The basic syntax for objects, messages and configurations is given by the following order-sorted rewrite signature, where the sorts OId, CId and AId are those of object, class and attribute identifiers respectively, and Msg is the sort of messages.

```
sorts Object Attribute Attributes Msg
  Configuration Value OId CId AId .
subsorts OId < Value .
subsorts Attribute < Attributes .
subsorts Object Msg < Configuration .
op <_:_|_> : OId CId Attributes -> Object .
op (_:_) : AId Value -> Attribute .
op _,_ : Attributes Attributes ->  Attributes
    [assoc comm id: nil] .
op __ : Configuration Configuration
     -> Configuration [assoc comm id: ∅] .
```

where the operators _,_ and __ are both associative and commutative with respective identities nil and $\emptyset$. 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) has the effect of constraining the attributes of objects in that class to be in a fixed set $\{a_1, \dots, a_n\}$ of attribute names, 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. 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; however, the above signature is sufficient for our present discussion.

As already mentioned, the *configuration* is the distributed state of the given concurrent object-oriented system and is represented as a multiset of objects and messages. The system evolves by concurrent rewriting (modulo ACI) of its configuration using rewrite rules that are specific to each particular system. The lefthand and righthand sides of such rules may in general involve patterns for several objects and messages. For example, objets in a class *Accnt* of bank accounts, each having a *bal*(ance) attribute, may receive messages for crediting or debiting the account and evolve according to the rules:

$$\begin{aligned} &credit(B, M)\ \langle B : Accnt \mid bal : N\rangle \\ &\quad\longrightarrow \langle B : Accnt \mid bal : N + M\rangle \end{aligned}$$

$$\begin{aligned} &debit(B, M)\ \langle B : Accnt \mid bal : N\rangle \\ &\quad\longrightarrow \langle B : Accnt \mid bal : N - M\rangle. \end{aligned}$$

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

$$\begin{aligned} (\dagger)\ & M_1 \dots M_n\ \langle O_1 : C_1 \mid attrs_1\rangle \dots \langle O_m : C_m \mid attrs_m\rangle \\ &\longrightarrow \langle O_{i_1} : C'_{i_1} \mid attrs'_{i_1}\rangle \dots \langle O_{i_k} : C'_{i_k} \mid attrs'_{i_k}\rangle \\ &\qquad \langle Q_1 : D_1 \mid attrs''_1\rangle \dots \langle Q_p : D_p \mid attrs''_p\rangle \\ &\qquad M'_1 \dots M'_q \end{aligned}$$

where the Ms are message expressions and $i_1, \dots, i_k$ are different numbers among the original $1, \dots, m$. 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 $M_1, \dots, M_n$ disappear;
- the *state* and possibly even the *class* of the objects $O_{i_1}, \dots, O_{i_k}$ may change;
- all other objects O_j vanish;
- new objects $Q_1, \dots, Q_p$ are created;
- new messages $M'_1, \dots, M'_q$ are sent.

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*[6]. 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 in principle 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 $O_1, \dots, O_m$ are all different, then the instances of the object names $Q_1, \dots, Q_m$ are also all different and different from the Os. Sufficient conditions to guarantee the uniqueness of objects are discussed in Section 4.3.

4.1 Object-Oriented Modules

Taking syntactic advantage of the structural properties common to all object-oriented systems, Maude allows the definition of *object-oriented modules* in addition to functional and system modules. However, the syntax and semantics of object-oriented modules can be reduced to that of system modules as explained in Section 4.2. The syntax of object-oriented modules is illustrated below by an example of a FIFO buffer of bounded size. The data elements to be stored in the buffer are a parameter; the other parameter is the size of the buffer which is specified by the parameter theory

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

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

```
omod BD-BUFF[X :: TRIV, K :: NAT*] is
  protecting LIST[X] .
  class BdBuff .
  att contents : BdBuff -> List [hidden] .
```

[6] In some cases, it may be convenient to think of the object name as the *pair* formed by its name and its class name; in this way, a greater reuse of object names is possible (see the example in Section 4.6.)

```
  msg put_in_ : Elt Id.BdBuff -> Msg.BdBuff .
  msg getfrom_replyto_ : Id.BdBuff OId
        -> Msg.BdBuff .
  msg elt-in_is_to_ : Id.BdBuff Elt OId -> Msg .
  var B : Id.BdBuff .
  var 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: E Q> =>
       <B : BdBuff | contents: Q>
       (elt-in B is E to I) .
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 (see Section 4.2.) The sort Id.BdBuff stands for object identifiers for the class BdBuff; similarly, Msg.BdBuff denotes messages that will participate in communication events involving only objects of class BdBuff. The two types of communication events of this kind that are possible are specified by the two rules of the module. The sort OId stands for identifiers of objects in *any* class. Therefore, if an arbitrary object I possesses the name B of a buffer, then it is possible for that object 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 (elt-in B is E to I).

Notice that this definition allows us to embed instances of the present module into much more complex object-oriented systems with complete flexibility for specifying *a posteriori* and *dynamically* which objects will communicate with buffers. In other words, Maude object modules support an "open systems" approach for defining complex object-oriented systems out of smaller subsystems.

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 obtain for the buffer; if additional error handling is desired, this can be specified by adding more rules. By contrast, a language like ABCL/1 [30] requires introducing a special "waiting mode" for objects and a corresponding "select construct" to reactivate the object appropriately after such waiting.

4.2 Reduction to System Modules

The syntax and semantics of a Maude object-oriented module are *entirely reducible to those of a system module*, i.e., we can systematically translate an object-oriented module omod $\mathcal{O}$ endom into a corresponding system module mod $\mathcal{O}\#$ endm whose semantics[7] is the object-oriented module's intended semantics.

The translation from a module omod $\mathcal{O}$ endom to a module mod $\mathcal{O}\#$ endm is illustrated by the following translated version of the bounded buffer module:

```
mod BD-BUFF#[X :: TRIV, K :: NAT*] is
  protecting LIST[X] .
  sorts BdBuff Id.BdBuff OId Msg.BdBuff Msg
    Value Configuration .
  subsorts Id.BdBuff < OId < Value .
  subsorts List < Value .
  subsorts Msg.BdBuff < Msg .
  subsorts Msg BdBuff < Configuration .
  op <_: BdBuff | contents:_> : Id.BdBuff List
       -> BdBuff .
  op put_in_ : Elt Id.BdBuff -> Msg.BdBuff .
  op getfrom_replyto_ : Id.BdBuff OId
       -> Msg.BdBuff .
  op elt-in_is_to_ : Id.BdBuff Elt OId -> Msg .
  op __ : Configuration Configuration
       -> Configuration [assoc comm id: Ø] .
  var B : Id.BdBuff .
  var 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: E Q> =>
       <B : BdBuff | contents: Q>
       (elt-in B is E to I) .
endm
```

Note that the rewrite rules are the same and therefore the behavior of the translated module is exactly the one we would expect from the original object-oriented module; what has been made more explicit is some of the order-sorted *type structure*, for which the conventions of object-oriented modules provide a form of syntactic sugar.

In general, a translation of this kind introduces for each *visible*, i.e., not hidden, attribute a of sort S for objects in a class C operators

```
op a._replyto_ : Id.C OId -> Msg.C .
op a._is_to_ : Id.C S OId -> Msg .
```

to create messages requesting that the value of the attribute of the recipient object is sent to another specified object; this behavior is specified by introducing the corresponding rule

```
rl (a. X reply to Y) <X : C | ATTS, a: V, ATTS'> =>
     <X : C | ATTS, a: V, ATTS'> (a. X is V to Y) .
```

The bounded buffer example does not exhibit such operators and rules because the only attribute, contents, was declared [hidden].

Note finally that, for the moment, the identifier sorts have not been specified further in order to leave open the issue of name conventions, which is touched upon in Section 4.3.

4.3 Object-Oriented Concurrency

Object synchronization, object creation, metaclasses and active objects, are discussed.

Synchrony and Asynchrony

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

[7] For the moment, consider the semantics of the module in terms of concurrent rewriting. Section 5 gives a model theoretic semantics for Maude modules that makes completely precise their intended semantics.

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* by the rule. For example, a *transfer* message to transfer funds between accounts could be defined in two versions (asynchronous and synchronous) as follows:

$$\mathit{asynch\text{-}transfer}(M, A, B) \longrightarrow \mathit{debit}(A, M)\ \mathit{credit}(B, M)$$

$$\begin{array}{l} \mathit{synch\text{-}transfer}(M, A, B)\ \langle A : \mathit{Accnt} \mid \mathit{bal} : N\rangle \\ \quad \langle B : \mathit{Accnt} \mid \mathit{bal} : N'\rangle \\ \qquad \longrightarrow \langle A : \mathit{Accnt} \mid \mathit{bal} : N - M\rangle\ \langle B : \mathit{Accnt} \mid \mathit{bal} : N' + M\rangle \end{array}$$

In the first case, the message will produce two asynchronous communication events —namely debiting to A and crediting to B the amount of money M— whereas in the second case both accounts have to be synchronized for the transaction to occur.

Object Creation and Metaclasses

Object creation and deletion can be treated in a simple way. In its simplest formulation, we can use numbers as object identifiers and a fixed *Counter* object, call it *Unique*, together with "*new*" messages for object creation and "*delete*" messages for object deletion with corresponding rules:

$$\begin{array}{l} \mathit{new}(C \mid \mathit{attrs})\ \langle \mathit{Unique} : \mathit{Counter} \mid \mathit{val} : N\rangle \\ \quad \longrightarrow \langle N + 1 : C \mid \mathit{attrs}\rangle\ \langle \mathit{Unique} : \mathit{Counter} \mid \mathit{val} : N + 1\rangle \end{array}$$

$$\mathit{delete}(A : C)\ \langle A : C \mid \mathit{attrs}\rangle \longrightarrow \emptyset$$

In a sense this makes unnecessary having new objects appear explicitly in the righthand sides of rules, since such objects can be replaced by corresponding *new* messages.

However, the above scheme is too crude, because the counter could easily become a bottleneck when many objects are being created; therefore, even if we want to use *new* messages to create objects, more flexible schemes should be used. The situation can be alleviated by organizing the creation of objects as a process that is *local to each class*. This can for example be achieved by introducing a class *Class* whose objects are representations of the existing classes and their corresponding objects. For each class C, the role of the natural numbers can then be played by a data type $\mathit{Id.C}$ having a function $\mathit{next} : \mathit{Id.C} \longrightarrow \mathit{Id.C}$ such that for each $I \in \mathit{Id.C}$, $n > 0$, $I \neq \mathit{next}^n(I)$. The object representing a class C can then be of the form

$$\langle C : \mathit{Class} \mid \mathit{last} : I, \mathit{objects} : L\rangle$$

where I is the last object identifier that was created and L is the list of the identifiers for existing objects of the class C. Our previous rules now take the form

$$\begin{array}{l} \mathit{new}(C \mid \mathit{attrs})\ \ \langle C : \mathit{Class} \mid \mathit{last} : I, \mathit{objects} : L\rangle \\ \quad \longrightarrow \langle \mathit{next}(I) : C \mid \mathit{attrs}\rangle \\ \qquad \langle C : \mathit{Class} \mid \mathit{last} : \mathit{next}(I), \mathit{objects} : \mathit{next}(I) L\rangle \end{array}$$

$$\begin{array}{l} \mathit{delete}_0(A : C)\ \ \langle C : \mathit{Class} \mid \mathit{last} : I, \mathit{objects} : L\rangle \\ \quad \longrightarrow \langle C : \mathit{Class} \mid \mathit{last} : I, \mathit{objects} : \mathit{remove}\ A\ \mathit{from}\ L\rangle \\ \qquad \mathit{delete}_1(A : C) \end{array}$$

$$\mathit{delete}_1(A : C)\ \ \langle A : C \mid \mathit{attrs}\rangle \longrightarrow \emptyset$$

where the deletion process first removes the object's identifier from the list of objects in the class and then deletes the object itself. A variant of the above scheme can be obtained by allowing "user specifiable" identifiers for objects, instead of using the function *next*, checking whether the name exists already and generating an appropriate error message in that case; it is even possible to combine both possibilities. The use of the class *Class* has also other obvious advantages. For example, by iterating on the list of current objects in a class it is very easy to *broadcast* a message to all of them.

Taking this approach a step further, we could also introduce a class *Mod* whose objects are representations of the modules in the current system, say

$$\langle M : \mathit{Mod} \mid \mathit{kind} : J, \mathit{signature} : S, \mathit{rules} : R, \mathit{submodules} : L\rangle$$

in such a way that extending the system with a new module definition could be accomplished by sending a *new* message specifying as attributes the kind, signature, rules and submodules of the module being defined. In case the module's kind is object-oriented, the effect of such a message should not only be that the module's name is appended to the *objects* attribute of the object *Mod* of class *Class* and that the object representing the new module is created; it should also have the subsequent effect of creating new objects of class *Class* for each one of the classes introduced in the module. This opens up interesting possibilities for reflection and metaprogramming in Maude.

There may be cases in which even the use of objects for representing classes could become a bottleneck for object creation. In such cases, one could adopt an *entirely distributed* approach to the creation of objects. A very easy expedient 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; it is even possible to send messages notifying the respective *Class* objects that the new objects exist. 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. The particular choice of mechanisms and the corresponding choice of data representations for object identifiers may depend on particular characteristics of the given application.

Active Objects

Messages, besides modifying one or more objects, can also cause other messages to be sent, thus initiating a possibly infinite chain reaction of messages. A simple example is provided by a *Clock* object with a *tick* message:

$$\begin{array}{l} \mathit{tick}(C)\ \ \langle C : \mathit{Clock} \mid \mathit{time} : T\rangle \\ \quad \longrightarrow \langle C : \mathit{Clock} \mid \mathit{time} : T + 1\rangle\ \ \mathit{tick}(C). \end{array}$$

Since this process never stops, it is reasonable to speak of the clock as an "active object." There is however an even more striking mode of activity, namely that of objects that, on their own, change their state and/or send messages to other objects *without any external prompting* by messages or by other objects. This is exemplified in Section 4.6 by philosophers that, all of a sudden, become hungry, try to pick up forks, etc. The key pattern permitting this kind of autonomous activity is that of rules whose lefthand sides involve just one object and no messages.

Since some of the attributes of an object —as well as the parameters of messages— can contain object names, very complex communication patterns can be achieved. Therefore, objects —besides being "active"— can *cooperate* in accomplishing very complex tasks.

4.4 Inheritance

Class inheritance is directly supported by Maude's order-sorted type structure. Therefore, a subclass declaration

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

Figure 3: A dictionary for Actors.

C < C' in an object-oriented module omod $\mathcal{O}$ endom is interpreted as a subsort declaration C < C' in its system module translation mod $\mathcal{O}\#$ endm. The effect in the signature of $\mathcal{O}\#$ is that the attributes of all the superclasses as well as the newly defined attributes of a subclass appear in the syntax definition of the constructor operator for objects in the subclass. Rules must also be inherited, unless new rules with a different behavior than those in a superclass are introduced for the subclass; at the level of the translation $\mathcal{O}\#$ this can be accomplished by introducing new rules that add extra attributes to the previous rules. Alternatively, if an *ACI* representation of the set of attributes is used, it is possible to turn rules associated with a class C into conditional rules that check whether the class identifier C' of the matched object is smaller than C in the subsort ordering.

4.5 Actors

Actors [2, 1] provide a very rich and interesting style of concurrent object-oriented programming. However, their mathematical structure, although already described and studied by previous researchers [6, 1], 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 and interesting approach.

Specifically, the general logical theory of concurrent objects presented in this paper yields directly 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 [2].

Actor theory has a terminology of its own which, to make things clearer, I will attempt to relate to the more standard terminology employed in object-oriented programming. To the best of my understanding, the table in Figure 3 provides a basic terminological correspondence of this kind.

The essential idea about actors is clearly summarized in the words of Agha and Hewitt [2] 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 form

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

Therefore, the present theory is considerably *more general* than that of actors; the dining philosophers example in Section 4.6 illustrates the advantages of this greater generality. In comparison with existing accounts about actors [2, 1] it seems also fair to say that our theory is *more abstract* so that some of those accounts can now be regarded as *high level architectural descriptions* of ways in which the abstract model 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. A nice feature of our approach is that it gives a *truly concurrent* formulation —in terms of concurrent *ACI*-rewriting (see Section 5 for more mathematical details)— of actor computations, which seems most natural given their character. By contrast, Agha [1] 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*."

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*. In the concurrent rewriting model it is possible to state precisely a variety of fairness conditions and, in particular, the guarantee of mail delivery for the special case of actors. However, space limitations preclude a treatment of fairness issues in this paper.

4.6 A Dining Philosophers Example

To further illustrate Maude's programming style, I include a dining philosophers example. The number k of philosophers is equal to the number of forks and is a parameter specified by the theory NAT*. I assume that the philosophers are seated in a round table, each occupying a fixed position. The circular geometry is captured abstractly by the natural numbers modulo k, which are defined as a parameterized functional module. Such numbers are used as identifiers for both philosophers and forks.

The code for the example introduces a useful convention that avoids making unnecessary mention of object attributes which are irrelevant for a given occurrence of an object in a rewrite rule. The general convention is that only a subset $\{a_1, \ldots, a_{n+k}\}$ of the attributes of (a pattern for) an object O may be mentioned in a righthand occurrence of the object in a given rule, and then a lefthand occurrence will mention a smaller or equal subset $\{a_1, \ldots, a_n\}$ of those attributes. What this abbreviates is a lefthand side pattern

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n, a_{n+1} : x_1, \ldots, a_{n+k} : x_k, attrs \rangle$$

where the x_js are new "don't care" variables, and *attrs* matches the remaining attributes, and a corresponding righthand side pattern

$$\langle O : C' \mid a_1 : v'_1, \ldots, a_{n+k} : v'_{n+k}, attrs \rangle.$$

The example illustrates the expressive power of the general form (†) of rules allowed in the logical theory of concurrent objects developed in this paper. Such rules permit dealing directly with phenomena that cannot be expressed naturally in the actor model due to the requirement that actors operate by reacting to messages that they receive. In the example, a philosopher can spontaneously pass from not being hungry —which we can implicitly interpret as a suitable state for thinking— to being hungry, but not yet restless. After some time, his hunger will be such that he will actually become restless and will attempt picking up the two forks adjacent to him. If the philosopher is in possession of the two forks for some time, he will satiate his appetite; therefore, by another spontaneous reaction, the philosopher will leave the two forks on the table. All those behavior instances occur *autonomously*, without the philosopher receiving any external communication, although communication events do also occur, at times involving receiving more than one message.

Of course, the example allows *deadlock* configurations. In rewriting terms this means configurations that *cannot be rewritten further* and that are undesirable in some intuitive or precisely specified sense. In this example, this can happen by configurations in which philosophers starve each other by keeping one of the forks. Agha [1] points out that one of the possibilities offered by the concurrent object-oriented programming style is to resolve deadlocks and other such conflicts by some kind of *cooperation* among objects, rather than by imposing a more traditional centralized way of resolving the conflict (such as a waiter in this example.) I present a particular solution of this kind, namely a community of wise philosophers, introduced in the subsequent parameterized module WISE-PHIL[K :: NAT*]. Each wise philosopher exercises *enlightened altruism* by releasing one of the forks when he finds out that the other fork is occupied. In this way, under appropriate fairness conditions, the philosophers can live peacefully together enjoying each other's company, without any worldly anxiety about their nourishment. Of course, other solutions —requiring fewer fairness assumptions— are also possible in the same decentralized style.

```
fmod NAT-MODULO[K :: NAT*] is
  sort Nat/k .
  op [_] : Nat -> Nat/k .
  var N : Nat .
  eq [N] = [N - k] if N >= k .
endfm

omod DINING-PHIL[K :: NAT*] is
  protecting  NAT-MODULO[K] .
  sort Answer .
  ops yes, no : -> Answer .
  classes Phil Fork .
  subsorts Nat/k < Id.Phil Id.Fork .
  atts lfork, rfork : Phil -> Answer .
  atts hungry, restless : Phil -> Answer [hidden] .
  att owner : Fork -> Id.Phil? .
  msg _picksup_ : Id.Phil Id.Fork -> Msg.Fork .
  msg _leaves_ : Id.Phil Id.Fork -> Msg.Fork .
  msg fork_for_ready_ : Id.Fork Id.Phil Answer
     -> Msg.Phil .
  vars N M : Nat .
  var O : Id.Phil? .
  rl <[N] : Phil | hungry: no> =>
       <[N] : Phil | hungry: yes, restless: no> .
  rl <[N] : Phil | lfork: no, rfork: no,
         hungry: yes, restless: no> =>
       <[N] : Phil | lfork: no, rfork: no,
         hungry: yes, restless: yes>
       ([N] picksup [N]) ([N] picksup [N + 1]) .
  rl <[N] : Phil | lfork: yes, rfork: yes,
           hungry: yes> =>
       <[N] : Phil | lfork: no, rfork: no,
         hungry: no, restless: no>
       ([N] leaves [N]) ([N] leaves [N + 1]) .
  rl (fork [N] for [M] ready yes)
       <[M] : Phil | hungry: yes> =>
       if [N] == [M]
       then <[M] : Phil | rfork: yes, hungry: yes>
       else
         if [N] == [M + 1]
         then <[M] : Phil | lfork: yes, hungry: yes>
         else <[M] : Phil | hungry: yes>
         fi
       fi .
  rl (fork [N] for [N] ready no)
       (fork [M] for [N] ready no)
       <[N] : Phil | lfork: no, rfork: no,
         hungry: yes, restless: yes> =>
       <[N] : Phil | lfork: no, rfork: no,
         hungry: yes, restless: no>
       if [N + 1] == [M] .
  rl ([N] picksup [M]) <[M] : Fork | owner: O> =>
       if (O == null and
         ([N] == [M] or [N + 1] == [M]))
       then <[M] : Fork | owner: [N]>
         (fork [M] for [N] ready yes)
       else <[M] : Fork | owner: O>
         (fork [M] for [N] ready no)
       fi .
  rl ([N] leaves [M])
       <[M] : Fork | owner: [N]> =>
       <[M] : Fork | owner: null>
endom

omod WISE-PHIL[K :: NAT*] is
  extending DINING-PHIL[K] .
  vars N M : Nat .
  rl (fork [N] for [N] ready no)
       <[N] : Phil | lfork: yes, rfork: no,
         hungry: yes> =>
       <[N] : Phil | lfork: no, rfork: no,
         hungry: yes, restless: no>
       ([N] leaves [N + 1]) .
  rl (fork [N] for [M] ready no)
       <[M] : Phil | lfork: no, rfork: yes,
         hungry: yes> =>
       <[M] : Phil | lfork: no, rfork: no,
         hungry: yes, restless: no>
       ([M] leaves [M + 1]) if [N] == [M + 1] .
endom
```

4.7 Architectural Considerations

As in any programming language that is truly based on logic, Maude modules have both a *declarative* and a *computational* reading. Indeed, their declarative reading is precisely a *formal specification* of the problem that they solve computationally. Since this paper has emphasized the logical and declarative reading of Maude programs (for more on this see Section 5) some remarks are in order to avoid overlooking the significance of their computational meaning for implementation purposes.

I should begin pointing out that my first work on concurrent rewriting, jointly with Joseph Goguen and Claude Kirchner [13], was precisely in the context of parallel architecture for the Rewrite Rule Machine [17], a massively parallel computer that we are building at SRI. The point is that concurrent rewriting is an ideal model of parallel computation that can be used as an abstract interface between very high level declarative languages and a parallel architecture that realizes physically such model.

Our original work in [13] concentrated on the case of *syntactic rewriting* which is the easiest to realize in hardware and directly supports many important computations. However, the subsequent paper [15] also gave implementation techniques for objects in the context of FOOPS. Such techniques did not make use of ACI-rewriting ideas; rather, they relied on particular restrictions or extensions of the syntactic rewriting model. The next natural step is to develop implementation techniques for concurrent object-oriented computation by means of rules of the general form (†). Notice that such rules make a very limited use of ACI-rewriting because no variables are ever used to match multisets; only individual elements of the configuration multiset are matched by each of the terms in the multiset expression making up the lefthand side. This suggests implementing directly such restricted forms of ACI-rewriting by means of *communication*. For example, a rule to credit money to an account can be implemented by identifying the name of the account with a particular *address* in the machine, and then routing the credit message to the location of its addressee so that when both become contiguous the rewrite rule for crediting the account can be applied. The development of general techniques of this kind is a very exciting research topic that I plan to pursue in the future.

5 Semantics

In this section I discuss models for rewriting logic and explain how such models are used to give semantics to modules in Maude. I focus on the basic ideas and intuitions and leave out some of the details, which can be found in [26, 25].

I 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 [21], with states as objects, transitions as morphisms, and sequential composition as morphism composition, and in them 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. Note that in the rest of this paper I 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 $f \in \Sigma_n$, $n \in \mathbb{N}$,
$$\frac{\alpha_1 : [t_1] \longrightarrow [t_1'] \quad \ldots \quad \alpha_n : [t_n] \longrightarrow [t_n']}{f(\alpha_1,\ldots,\alpha_n) : [f(t_1,\ldots,t_n)] \longrightarrow [f(t_1',\ldots,t_n')]}$$

3. **Replacement.** For each rewrite rule $r : [t(\overline{x}^n)] \longrightarrow [t'(\overline{x}^n)]$ in R,
$$\frac{\alpha_1 : [w_1] \longrightarrow [w_1'] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w_n']}{r(\alpha_1,\ldots,\alpha_n) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}$$

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*. I will always 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, I 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 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[8]:

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 $\alpha_1,\ldots,\alpha_n,\beta_1,\ldots,\beta_n$,
   $$f(\alpha_1;\beta_1,\ldots,\alpha_n;\beta_n) = f(\alpha_1,\ldots,\alpha_n);f(\beta_1,\ldots,\beta_n)$$
   (b) *Preservation of identities.*
   $$f([t_1],\ldots,[t_n]) = [f(t_1,\ldots,t_n)]$$

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 $X = \emptyset$, I 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 rule— 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 matched subterms. The exchange law also states that rewriting at the top by means of rule r and rewriting "below" are processes that are independent of each other and therefore can be done in any order. Therefore, $r(\overline{\alpha})$ is also equivalent to the sequential composition $t(\overline{\alpha});r(\overline{[w']})$. Since $[t(x_1,\ldots,x_n)]$ and $[t'(x_1,\ldots,x_n)]$ can be regarded as functors $\mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$, the exchange law just asserts that r is a *natural transformation* [21], i.e.,

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

Lemma 3 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 step sequences or as interleaving sequences:

Lemma 4 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. □

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

Definition 5 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, i.e., for each $f \in \Sigma_n$, $n \in \mathbb{N}$, a functor $f_{\mathcal{S}} : \mathcal{S}^n \longrightarrow \mathcal{S}$, in such a way that the equations E are satisfied, 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 $r_{\mathcal{S}} * F = F^n * r_{\mathcal{S}'}$, 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" I 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 (see [26, 25].) 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 [27, 28]; 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 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.

System	⟷	*Category*
State	⟷	*Object*
Transition	⟷	*Morphism*
Procedure	⟷	*Natural Transformation*
Distributed Structure	⟷	*Algebraic Structure*

Figure 4: The mathematical structure of concurrent systems.

The expressive power of rewrite theories to specify concurrent transition systems[9] 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*[10] 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 4 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 [25], where the soundness and completeness of rewriting logic for $\mathcal{R}$-system models is also proved. Below, for $\mathcal{C}$ a category, $Obj(\mathcal{C})$ denotes the set of its objects.

Theorem 6 $\mathcal{T}_{\mathcal{R}}$ is an initial object in the category $\underline{\mathcal{R}\text{-}Sys}$. More generally, $\mathcal{T}_{\mathcal{R}}(X)$ has the following universal property:

[9] 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 [25].

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

Given an $\mathcal{R}$-system $\mathcal{S}$, each function $F : X \longrightarrow Obj(\mathcal{S})$ extends uniquely to an $\mathcal{R}$-homomorphism $F^{\natural} : \mathcal{T}_{\mathcal{R}}(X) \longrightarrow \mathcal{S}$. □

5.1 Preorder, Poset and Algebra Models

Since $\mathcal{R}$-systems are an "essentially algebraic" concept[11], we can consider classes Θ of $\mathcal{R}$-systems defined by the satisfaction of additional equations. Such classes give rise to full subcategory inclusions $\Theta \hookrightarrow \underline{\mathcal{R}\text{-}Sys}$, and by general universal algebra results about essentially algebraic theories (see, e.g., [3]) such inclusions are *reflective* [21], 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 $F^{\diamond} : R_{\Theta}(\mathcal{S}) \longrightarrow \mathcal{D}$ such that $F = \rho_{\Theta}(\mathcal{S}); F^{\diamond}$.

Therefore, we can consider subcategories of $\underline{\mathcal{R}\text{-}Sys}$ that are defined by certain equations and be guaranteed that they have initial and free objects, that they are closed by subobjects and products, etc. Consider for example the following equations:

$$\forall f, g \in Arrows,\ f = g \ \text{ if } \ \partial_0(f) = \partial_0(g) \wedge \partial_1(f) = \partial_1(g)$$
$$\forall f, g \in Arrows,\ f = g \ \text{ if } \ \partial_0(f) = \partial_1(g) \wedge \partial_1(f) = \partial_0(g)$$
$$\forall f \in Arrows,\ \partial_0(f) = \partial_1(f).$$

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

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

A routine inspection of $\underline{\mathcal{R}\text{-}Preord}$ 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, $\underline{\mathcal{R}\text{-}Alg}$ is the category of ordinary Σ-algebras that satisfy the equations $E \cup unlabel(R)$, where the *unlabel* function removes the labels from the rules and turns the sequent signs " $\longrightarrow$ " into equality signs.

The reflection functor associated to the inclusion $\underline{\mathcal{R}\text{-}Preord} \subseteq \underline{\mathcal{R}\text{-}Sys}$, sends $\mathcal{T}_{\mathcal{R}}(X)$ to the familiar $\mathcal{R}$-*rewriting relation*[12] $\rightarrow_{\mathcal{R}(X)}$ on E-equivalence classes of terms with variables in X. Similarly, the reflection associated to the inclusion $\underline{\mathcal{R}\text{-}Pos} \subseteq \underline{\mathcal{R}\text{-}Sys}$ 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 $\underline{\mathcal{R}\text{-}Alg}$ maps $\mathcal{T}_{\mathcal{R}}(X)$ to $T_{\mathcal{R}}(X)$, the free Σ-algebra on X satisfying the equations $E \cup unlabel(R)$; therefore, the classical *initial algebra semantics* of (functional) equational specifications reappears here associated to a very special class of models which —when viewed as systems— have only trivial identity transitions.

5.2 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. I 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[13] 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.

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 $\mathcal{R}'$ rewriting theories[14]. 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.

Definition 7 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 $\mathcal{T}_{\mathcal{R}} \longrightarrow R_{\Theta}(\mathcal{T}_{\mathcal{R}})$ □

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 $\mathcal{S} = \mathcal{T}_{\mathtt{NAT}}$, for `NAT` the rewriting theory of the Peano natural numbers corresponding to the module `NAT`[15] in Section 2, with $\mathcal{M} = \mathbb{N}$, and with $[\![\_]\!]$ the unique homomorphism from the initial `NAT`-system $\mathcal{T}_{\mathtt{NAT}}$; i.e., this is the initial machine in $\underline{\mathtt{NAT}\text{-}Mach/\mathtt{NAT}\text{-}Alg}$. 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*[16] `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*

[11] In the precise sense of being specifiable by an "essentially algebraic theory" or a "sketch" [3]; see [25].

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

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

[14] This is somewhat inaccurate in the case of system modules having functional submodules, which is treated in [26], because we have to "remember" that the submodule in question is functional.

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

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

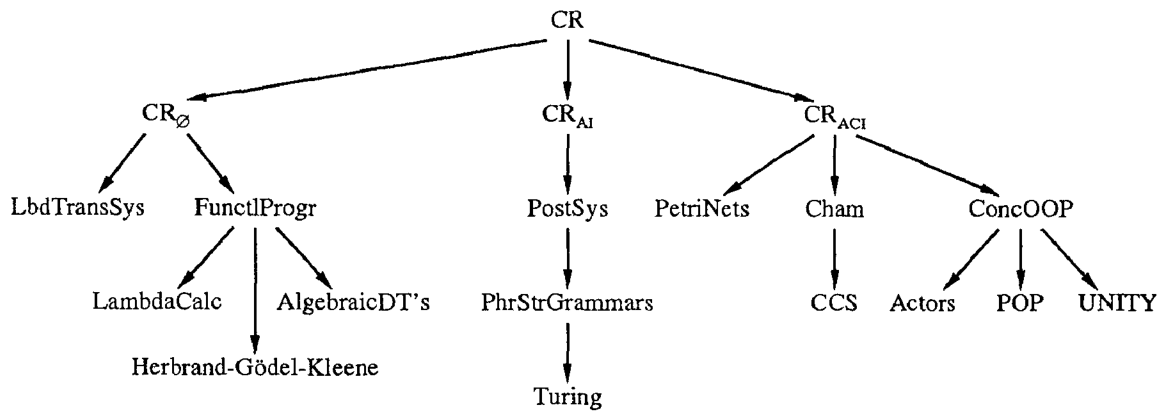


Figure 5: Unification of Concurrency Models.

mod $\mathcal{R}$ endm not having any functional submodules[17] I propose the initial machine in $\mathcal{R}$-*Mach*/$\mathcal{R}$-*Preord*, but other choices are also possible. On the one hand, we could choose to be as concrete as possible and take $\Theta = \mathcal{R}$-*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 $\Theta = \mathcal{R}$-*Pos*; however, this would have the unfortunate effect of collapsing all the states of a cyclic rewriting, which seems undesirable 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. Note that this makes Maude a *logic programming language* in the general axiomatic sense of [24].

In Maude a module can have *submodules*. Functional modules can only have functional submodules, system modules can have both functional and system submodules[18]. For example, NAT was declared a submodule of NAT-CHOICE. The meaning of submodule relations in which the submodule and the supermodule are both of the same kind is the obvious one, i.e., we augment the signature, equations, labels, and rules of the submodule by adding to them the corresponding ones in the supermodule; we then give semantics to the module so obtained according to its kind, i.e., functional or system. The semantics of a system module having a functional submodule is somewhat more delicate; this case is treated in [26].

As OBJ3, Maude has also *theories* to specify semantic requirements for interfaces and to make high level assertions about modules; they can be functional, system, or object-oriented; the examples in this paper have only used functional theories. Also as OBJ, Maude has *parameterized modules* —again of the three kinds— and *views* that are theory interpretations relating theories to modules or to other theories. Details regarding the semantics of all these aspects of the language will appear elsewhere[19].

6 Related Work and Concluding Remarks

Within the space constraints of this paper it is impossible to do justice to the wealth of related literature on concurrent object-oriented programming, term rewriting, abstract data types, concurrency theory, Petri nets, linear and equational logic, ordered, continuous and nondeterministic algebras, etc. A lengthier report [25] contains 85 such references. I would like to discuss the relationships of Maude with the FOOPS language that Joseph Goguen and I developed in [14], further studied with Ellen Munthe-Kaas (unpublished,) and that more recently has been further developed and enriched with new semantic ideas by Joseph Goguen and his coworkers at Oxford University [11, 10, 19].

Besides having provided a very valuable stimulus for Maude, FOOPS has also the nice common feature of including OBJ3 as its functional sublanguage. The main differences are in their semantics —with Maude's based on rewriting logic as explained in this paper, and FOOPS having received a reflective semantics [14], a direct algebraic semantics [14, 11], and a sheaf semantics [10]— and also in their computational and linguistic primitives —FOOPS based on methods and method expressions, Maude based on a paradigm of communication by messages— and in their different treatment of concurrency that is more limited in FOOPS. Regarding operational semantics and implementation ideas, there are also commonalities. In fact, the idea of transforming objects by rewrite rules goes back to [15], although the use of *ACI* to treat concurrency was not contemplated in that work, and inter-object communication and object creation and deletion were not explicitly addressed.

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 [25, 26]. However, we can summarize such specializations using Figure 5, 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 transitions systems, which are used in interleaving approaches to concurrency; functional programming (in particular Maude's functional modules) corresponds to the case of *confluent*[20] rules, and includes the λ-calculus (in combinator form) and the Herbrand-Gödel-Kleene theory of recursive functions. Rewriting modulo *AI* yields Post systems and related grammar formalisms, including Turing machines. Besides concurrent object-oriented programming, rewriting modulo *ACI* includes Berry and Boudol's *chemical abstract machine* [4] (which itself specializes to CCS [29]),

[17] See below for a discussion of submodule issues.

[18] Object-oriented modules can have submodules of the three kinds, but after reducing object-oriented modules to system modules no new issues appear for them.

[19] Some basic results about views and parameterization for system modules have already been given in [25].

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

as well as Unity's model of computation [5]; another special case is Engelfriet et al.'s POPs and POTs higher level Petri nets for actors [8, 9].

In summary, we have seen how a logical semantics for concurrent objects can be naturally obtained by regarding a concurrent object-oriented system as a rewrite theory. Concurrent object-oriented programming has in this way been transformed into a form of *logic programming* in the general axiomatic sense of [24]; this programming style has been illustrated with examples in Maude, a logic programming language containing OBJ3 as a functional sublanguage that unifies functional programming and concurrent object-oriented programming. Besides reducing computation to deduction, a *model theoretic semantics* for Maude has also been given.

References

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

[2] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming.* MIT Press, 1988.

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

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

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

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

[7] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Vol. B.* North-Holland, 1990.

[8] J. Engelfriet Net-based description of parallel object-based systems, or POTs and POPs. Technical report, Noordwijkerhout FOOL Workshop, May 1990.

[9] J. Engelfriet, G. Leih, and G. Rozenberg. Parallel object-based systems and Petri nets, I and II. Technical Report 90-04-5, Dept. of Computer Science, University of Leiden, February 1990.

[10] Joseph Goguen. Sheaf semantics for concurrent interacting objects. Manuscript, University of Oxford, May 1990; given as a tutorial at the REX School on Foundations of Object-Oriented Programming, Noorwijkerhout, The Netherlands, May 28-June 1, 1990.

[11] Joseph Goguen. Types as theories. Manuscript, University of Oxford, March 1990; to appear in G.M. Reed, A.W. Roscoe and R. Wachter (eds.), *Proceedings of the Oxford Symposium on Topology in Computer Science*, Oxford University Press, 1990.

[12] 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-Verlag, Lecture Notes in Computer Science No. 308, 1988.

[13] 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.

[14] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986; also, Technical Report CSLI-87-93, Center for the Study of Language and Information, Stanford University, March 1987.

[15] 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.

[16] Joseph Goguen and José Meseguer. Order-sorted algebra I: Partial and overloaded operations, errors and inheritance. Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989. Given as lecture at Seminar on Types, Carnegie-Mellon University, June 1983. Submitted for publication.

[17] 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 Lab, March 1989.

[18] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.

[19] Joseph A. Goguen and David Wolfram. On types and FOOPS. Manuscript, University of Oxford, 1990; to appear in Proc. IFIP TC-2 Conf. on Object-Oriented Databases, Windemere.

[20] 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.

[21] Saunders MacLane. *Categories for the working mathematician.* Springer, 1971.

[22] Narciso Martí-Oliet and José Meseguer. An algebraic axiomatization of linear logic models. Technical Report SRI-CSL-89-11, SRI International, Computer Science Lab, December 1989. To appear in G.M. Reed, A.W. Roscoe and R. Wachter (eds.), *Proceedings of the Oxford Symposium on Topology in Computer Science*, Oxford University Press, 1990.

[23] Narciso Martí-Oliet and José Meseguer. From Petri nets to linear logic. In D.H. Pitt et al., editor, *Category Theory and Computer Science*, pages 313–340. Springer Lecture Notes in Computer Science, Vol. 389, 1989. Full version to appear in *Mathematical Structures in Computer Science.*

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

[25] 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.

[26] José Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of the Concur'90 Conference, Amsterdam, August 1990.* Springer LNCS, 1990.

[27] José Meseguer and Ugo Montanari. Petri nets are monoids. Technical report, SRI International, Computer Science Laboratory, January 1988. Revised June 1989; to appear in *Information and Computation.*

[28] José Meseguer and Ugo Montanari. Petri nets are monoids: A new algebraic foundation for net theory. In *Proc. LICS'88*, pages 155–164. IEEE, 1988.

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

[30] 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.