Rewriting as a Unified Model of Concurrency*

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

1 Introduction

The main goal of this paper is to propose a general and precise answer to the question:

What is a concurrent system?

It seems fair to say that this question has not yet received a satisfactory answer, and that the resulting situation is one of *conceptual fragmentation* within the field of concurrency. A related problem is the *integration of concurrent programming with other programming paradigms*, such as functional and object-oriented programming. Integration attempts typically graft an existing concurrency model on top of an existing language, but such *ad hoc* combinations often lead to monstrous deformities which are extremely difficult to understand. Instead, this paper proposes a *semantic integration* of those paradigms based on a common *logic and model theory*.

The logic, called *rewriting logic*, is implicit in term rewriting systems but has passed for the most part unnoticed due to our overwhelming tendency to associate term rewriting with equational logic. Its proof theory exactly corresponds to (truly) concurrent computation, and the model theory proposed for it in this paper provides the general concept of concurrent system that we are seeking.

This paper also proposes rewrite rules as a very high level language to program concurrent systems. Specifically, a language design based on rewriting logic is presented containing a *functional sublanguage* entirely similar to OBJ3 [10] as well as more general *system modules*, and also *object-oriented modules* that provide notational convenience for object-oriented applications but are reducible to system modules [24]. The language's semantics is directly based on the model theory of rewriting logic and yields the desired semantic integration of concurrency with functional and object-oriented programming.

The resulting notion of concurrent system is indeed very general and specializes to a wide variety of existing notions in a very natural way. Section 5 discusses the specializations to labelled transition systems, Petri nets and concurrent object-oriented programming in some detail, and summarizes several others; however, space limitations preclude a more comprehensive discussion, for which we refer the reader to [25].

Acknowledgements. I specially thank Prof. Joseph Goguen for our long term collaboration on the OBJ and FOOPS languages [10, 11], concurrent rewriting [15] and its implementation on the RRM architecture [9, 13], all of which have directly influenced this work; he has also provided many positive suggestions for improving a previous version of this paper. I specially thank Prof. Ugo Montanari for our collaboration on the semantics of Petri nets [26, 27]; the algebraic ideas that we developed in that work have 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 [22, 21], which is another source of

*Supported by Office of Naval Research Contracts N00014-90-C-0086, N00014-88-C-0618 and N00014-86-C-0450, and NSF Grant CCR-8707155.

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 Dr. Claude Kirchner, Dr. Sany Leinwand, and 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. I also wish to thank Prof. Pierre-Louis Curien, Prof. Pierpaolo Degano, Prof. Brian Mayoh, Prof. Robin Milner, Dr. Mark Moriconi, Prof. Peter Mosses and Dr. Axel Poigné, all of whom provided helpful comments and encouragement.

2 Concurrent Rewriting

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 `NAT` module written in a notation similar to that of OBJ:

```
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
```

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

`NAT` defines the Peano natural numbers. It begins with the keyword `fmod` followed by the module's name, and ends with the keyword `endfm`. The sort `Nat` is declared using the keyword `sort`. 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," (for example, `s_`), "infix" (`_+_`) or any "mixfix" combination as well as standard parenthesized notation. Variables to be used for defining equations are declared with their corresponding sorts, and then equations are given (in this example, equations for addition); such equations provide the actual "code" of the module.

To compute with this module, 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.

The above module has equations that are Church-Rosser and terminating and is therefore *functional*; its mathematical semantics is given by the *initial algebra*[2] associated to the syntax and equations in the module [14], 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. Indeed,

[1] In OBJ it is possible to declare several attributes of this kind for an operator, including also associativity and identity, and then do rewriting modulo such attributes.

[2] In the above example, the initial algebra is of course the natural numbers with successor and addition.

Figure 1: Concurrent rewriting of an arithmetic expression

most of the applications discussed in this paper, including many well known concurrency models, are not functional. Consider the `NAT-CHOICE` module above, which adds a nondeterministic choice operator to the natural numbers. 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 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 are not identified by the new equations introduced in `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, I have used the keyword `mod` (instead of the previous `fmod`) to indicate that this module *is not functional*. Similarly, I have written a new keyword `rl` —instead of the usual `eq` before each equation— and replaced the equal sign 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.

2.1 Basic Universal Algebra

For the sake of making the exposition simpler, 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 $t(\overline{u}/\overline{x})$.

2.2 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., [17, 5].)

Given a signature (Σ, E), the *sentences* 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 (see Section 5.)

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

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

[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 3 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, in this paper I consider only *unconditional* rewrite rules. However, all the results presented here have been extended to conditional rules in [25] with very general rules of the form

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

This of course increases considerably the expressive power of rewrite theories.

[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.

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.**

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

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.

2.3 Concurrent Rewriting

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 was 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 $[t] \longrightarrow [t']$. $\Box$

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.

3 Semantics

As such, a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ is a *static description* of what a system can do. The *meaning* should be given by a model of its actual *behavior*. I construct below a model in which 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 the equivalence classes of terms $[t] \in T_{\Sigma,E}(X)$ and whose morphisms are equivalence classes of terms representing proofs in rewriting deduction, i.e., concurrent $\mathcal{R}$-rewrites. The rules for generating such terms, with the specification of their respective domain and codomain, are given below. 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 $f \in \Sigma_n$, $n \in \mathbb{N}$,

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

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

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

4. **Composition.**

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

For the case of equational logic we can also define a similar model as a category $\mathcal{T}^{\leftrightarrow}_{\mathcal{R}}(X)$ (actually a *groupoid*[6]) by using the rule of symmetry to generate additional terms:

5. **Inversion.**

$$\frac{\alpha : [t_1] \longrightarrow [t_2]}{\alpha^{-1} : [t_2] \longrightarrow [t_1]}$$

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[7]:

[6] A category $\mathcal{C}$ is called a *groupoid* iff any morphism $f : A \longrightarrow B$ in $\mathcal{C}$ has an inverse morphism $f^{-1} : B \longrightarrow A$ such that $f;f^{-1} = 1_A$, and $f^{-1};f = 1_B$.

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

1. **Category.**
   (a) *Associativity.* For all α, β, γ $\quad (\alpha; \beta); \gamma = \alpha; (\beta; \gamma)$
   (b) *Identities.* For each $\alpha : [t] \longrightarrow [t']$ $\quad \alpha; [t'] = \alpha$ *and* $[t]; \alpha = \alpha$
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.* $\quad f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$
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,

$$\frac{\alpha_1 : [w_1] \longrightarrow [w'_1] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w'_n]}{r(\overline{\alpha}) = r(\overline{[w]}); t'(\overline{\alpha}) = t(\overline{\alpha}); r(\overline{[w']})}$$

Similarly, the groupoid $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}(X)$ is obtained by identifying the terms generated by rules 1-5 modulo the above equations plus the additional:

5. **Inverse.** For any $\alpha : [t] \longrightarrow [t']$ in $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}(X)$, $\quad \alpha; \alpha^{-1} = [t]$ *and* $\alpha^{-1}; \alpha = [t']$

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 notations $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}$, respectively. 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* [20], i.e.,

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* view of the computations of the rewrite theory $\mathcal{R}$ that can reasonably be given. In particular, we can prove that all 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.

An *$\mathcal{R}$-groupoid* is an $\mathcal{R}$-system $\mathcal{S}$ whose category structure is actually a groupoid. This defines a full subcategory $\underline{\mathcal{R}\text{-}Grpd} \subseteq \underline{\mathcal{R}\text{-}Sys}$. □

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, this is in a sense the end of the story (see Section 5.1.) As I will argue and justify more fully with examples in Section 5, 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 [26, 27]. 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. 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 and, more generally, in any type of concurrent rewriting.

The expressive power of rewrite theories to specify concurrent transition systems[8] is greatly increased by the possibility of having not only transitions, but also *parameterized transitions*,

[8]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].

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

Figure 2: The mathematical structure of concurrent systems

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*[9] 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 2 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.

Theorem 6 $\mathcal{T}_{\mathcal{R}}$ is an initial object in the category $\underline{\mathcal{R}\text{-}Sys}$, and $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}$ is an initial object in the category $\underline{\mathcal{R}\text{-}Grpd}$. More generally, $\mathcal{T}_{\mathcal{R}}(X)$ has the following universal property: Given an $\mathcal{R}$-system $\mathcal{S}$, each function $F : X \longrightarrow Obj(\mathcal{S})$ extends uniquely to an $\mathcal{R}$-homomorphism $F^{\natural} : \mathcal{T}_{\mathcal{R}}(X) \longrightarrow \mathcal{S}$. $\mathcal{T}_{\mathcal{R}}^{\leftrightarrow}(X)$ has the same universal property with respect to $\mathcal{R}$-groupoids. □

3.1 Equationally Defined Classes of Models

Since $\mathcal{R}$-systems are an "essentially algebraic" concept[10], 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., [2]) such inclusions are *reflective* [20], i.e., for each $\mathcal{R}$-system $\mathcal{S}$ there is an $\mathcal{R}$-system $R_\Theta(\mathcal{S}) \in \Theta$ and an $\mathcal{R}$-homomorphism $\rho_\Theta(\mathcal{S}) : \mathcal{S} \longrightarrow R_\Theta(\mathcal{S})$ such that for any $\mathcal{R}$-homomorphism $F : \mathcal{S} \longrightarrow \mathcal{D}$ with $\mathcal{D} \in \Theta$ there is a unique $\mathcal{R}$-homomorphism $F^{\diamondsuit} : R_\Theta(\mathcal{S}) \longrightarrow \mathcal{D}$ such that $F = \rho_\Theta(\mathcal{S}); F^{\diamondsuit}$. The full subcategory $\underline{\mathcal{R}\text{-}Grpd} \subseteq \underline{\mathcal{R}\text{-}Sys}$ is also reflective, but it is not equationally definable. The situation generalizes that of the inclusion of the category of groups into the category of monoids. What we have in this case is an inclusion that is a *forgetful functor* from a category of algebras with additional operations (in this case the inversion operation.) However, for any equationally definable (full) subcategory $\Theta \subseteq \underline{\mathcal{R}\text{-}Sys}$, defined by a collection of equations H, the intersection $\Theta \cap \underline{\mathcal{R}\text{-}Grpd}$ has a very simple description, since it is just the full subcategory of $\underline{\mathcal{R}\text{-}Grpd}$ definable by the equations H.

Therefore, we can consider subcategories of $\underline{\mathcal{R}\text{-}Sys}$ or of $\underline{\mathcal{R}\text{-}Grpd}$ 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 conditional equations:

$$\forall f, g \in Arrows,\ f = g\ \ if\ \ \partial_0(f) = \partial_0(g) \wedge \partial_1(f) = \partial_1(g)$$

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

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

[10] In the precise sense of being specifiable by an "essentially algebraic theory" or a "sketch" [2]; see [25].

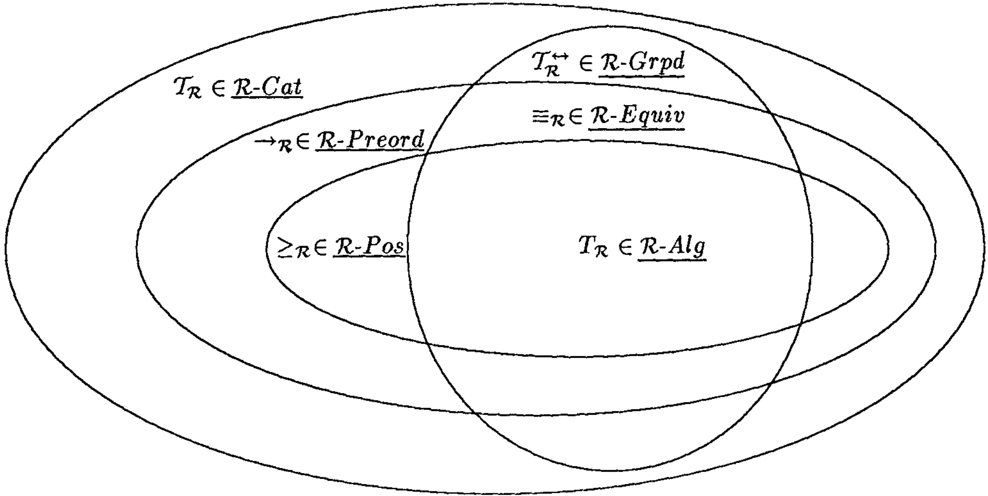


Figure 3: Subcategories of $\underline{\mathcal{R}\text{-}Sys}$ and their initial objects

The first equation forces a category to be a preorder, and the addition of the second requires this preorder to be a poset. By imposing the first one, or by imposing both, we get full subcategories

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

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. 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*[11] $\rightarrow_{\mathcal{R}(X)}$ on E-equivalence classes of terms with variables in X. It is easy to show that rewriting logic remains complete when we restrict the models to be preorders [25]. 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 $[t'] \rightarrow_{\mathcal{R}(X)} [t]$. Again, rewriting logic remains complete for poset models [25].

Intersecting $\underline{\mathcal{R}\text{-}Pos}$ and $\underline{\mathcal{R}\text{-}Preord}$ with the category $\underline{\mathcal{R}\text{-}Grpd}$ we get two subcategories definable by the first equation or by both, but now in the context of $\underline{\mathcal{R}\text{-}Grpd}$. Combining the notions of a groupoid and a preorder we get exactly the notion of an *equivalence relation* and therefore a subcategory $\underline{\mathcal{R}\text{-}Equiv}$ whose initial object is the usual congruence $\equiv_\mathcal{R}$ on ground terms modulo provable equality generated by the rules in $\mathcal{R}$ when regarded as equations. A poset that is also a groupoid yields a *discrete category* whose only arrows are identities, i.e., a set. It is therefore easy to see that the subcategory obtained by intersecting $\underline{\mathcal{R}\text{-}Pos}$ with $\underline{\mathcal{R}\text{-}Grpd}$ is just the familiar category $\underline{\mathcal{R}\text{-}Alg}$ 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. Similarly, 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. Figure 3 summarizes the relationships among all these categories.

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

4 Rewrite Rules as a Programming Language

In this paper I have put forward the view 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*— with the same high standards of mathematical rigor for its semantics. I present below a specific proposal for such a semantics. This proposal has two advantages. First, the functional case of equational logic is kept as a sublanguage having a more specialized semantics; second, the operational and mathematical semantics of a module are related in a particularly nice way. The proposal is embodied in Maude, a language design that contains OBJ3 [10] as its functional sublanguage. As already mentioned, all the ideas and results in this paper extend without problem[12] to the *order-sorted* case[13]; 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. In Maude there are three kinds of *modules*: *functional* —introduced by the keyword `fmod`, such as the `NAT` module in Section 2—, *system* —introduced by the keyword `mod` such as the module `NAT-CHOICE`— and *object-oriented* —introduced by the keyword `omod` (See Section 5.3.) The semantics of object-oriented modules reduces to that of system modules; therefore, in this section we focus on the functional and system cases. Functional and system modules 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 a *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* `mod` $\mathcal{R}$ `endm` not having any functional submodules[17] I propose the initial machine in $\underline{\mathcal{R}\text{-}Mach/\mathcal{R}\text{-}Preord}$, but other choices are also possible. On the one hand, we could choose to be as concrete as possible and take $\Theta = \underline{\mathcal{R}\text{-}Sys}$ in which case the abstraction map is the identity homomorphism for $\mathcal{T}_{\mathcal{R}}$. On the other hand, we could instead be even more abstract, and choose

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

[13] I.e., there is not just one sort, but a partially ordered set of sorts —with the ordering understood as type inclusion— and the function symbols can be overloaded [12].

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

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

$\Theta = \underline{\mathcal{R}\text{-}Pos}$; 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*.

In Maude a module can have *submodules*. Functional modules can only have functional submodules, but system modules can have both functional and system submodules. 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. Suppose that the rewrite theory of the functional submodule[18] is $\mathcal{R} = (\Sigma, E, L, R)$ and that of the system supermodule plus its system submodules is $\mathcal{R}' = (\Sigma', E', L', R')$; as before we can form $\mathcal{R} \cup \mathcal{R}' = (\Sigma \cup \Sigma', E \cup E', L \cup L', R \cup R')$, but the semantics of the module is now given by the initial machine in the category

$$\underline{(\mathcal{R} \cup \mathcal{R}')\text{-}Mach/(\Sigma \cup \Sigma', E \cup E' \cup unlabel(R), L', R')\text{-}Preord}.$$

Notice that $\underline{(\Sigma \cup \Sigma', E \cup E' \cup unlabel(R), L', R')\text{-}Preord}$ is an equationally definable full subcategory of $\underline{(\mathcal{R} \cup \mathcal{R}')\text{-}Preord}$, namely the one defined by the equations $t(\overline{x}) = t'(\overline{x})$ for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$, and therefore is also reflective.

Given a preorder $\mathcal{M}$ in $\underline{(\Sigma \cup \Sigma', E \cup E' \cup unlabel(R), L', R')\text{-}Preord}$ we can forget about R' and the labels and view it as an $\mathcal{R}$-algebra $\mathcal{M}|_{\mathcal{R}}$. Given a system module `mod` $\mathcal{R}'$ `endm` having `fmod` $\mathcal{R}$ `endfm` as its functional submodule and $\mathcal{T}_{\mathcal{R}\cup\mathcal{R}'} \longrightarrow \mathcal{M}$ as its semantics, we say that this submodule relation is *extending* if the unique $\mathcal{R}$-homomorphism $h : \mathcal{T}_{\mathcal{R}} \longrightarrow \mathcal{M}|_{\mathcal{R}}$ is injective; similarly, we say that it is *protecting* if h is an isomorphism. We leave for the reader to check that the extending relation asserted for the importation of `NAT` in `NAT-CHOICE` does in fact hold.

As OBJ, 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. 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 for all these aspects of the language will appear elsewhere[19]. Finally, note that Maude is a *logic programming language* in the general axiomatic sense made precise in [23].

5 Unifying Models of Concurrency

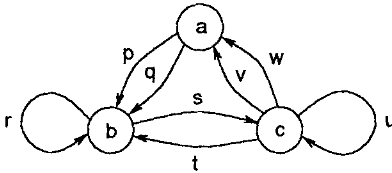
Labelled transition systems, Petri nets and concurrent object-oriented programming are discussed as specializations of concurrent rewriting; other specializations are also discussed briefly.

5.1 Labelled Transition Systems

This is the particularly simple case of rewrite theories $\mathcal{R} = (\Sigma, E, L, R)$ such that $E = \emptyset$, $\Sigma = \Sigma_0$, i.e., Σ only involves constants, and all the rules in $\mathcal{R}$ are of the form $r : a \longrightarrow b$ for a, b constants. For example, the transition system of Figure 4 corresponds to the rewrite theory of the sytem module `LTS` in the same figure. Since Σ contains only constants and the rules have no variables, the rules 1-5 of rewriting logic specialize to very simple rules. The rule of congruence becomes a trivial subcase of reflexivity, and the rule of replacement just yields each rule $r : a \longrightarrow b$ in $\mathcal{R}$ as its own consequence. Thus, we just have reflexivity and transitivity with

[18] We assume that, if several functional submodules have been declared, we have already taken their union.

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




```
mod LTS is
  sort State .
  ops a,b,c : -> State .
  rls p,q : a => b .
  rl    r : b => b .
  rl    s : b => c .
  rls v,w : c => a .
  rl    t : c => b .
  rl    u : c => c .
endm
```

Figure 4: A labelled transition system and its code in Maude

the rules $r : a \longrightarrow b$ in $\mathcal{R}$ as basic axioms. Therefore, $\mathcal{T}_{\mathcal{R}}$ is just the *free category* —also called the *path category*— generated by the labelled transition system when regarded as a graph. More generally, *any* $\mathcal{R}$-system with $\mathcal{R}$ a labelled transition system is just a *category* $\mathcal{C}$ together with the assignment of an object of $\mathcal{C}$ to each constant in Σ and a morphism in $\mathcal{C}$ for each rule in R in a way consistent with the assignment of objects. In other words, such systems are just *sequential systems*, and their sequentiality is precisely due to the absence of any operations other than constants. In fact, labelled transition systems are intrinsically *sequential* as rewrite theories, in the precise sense of Definition 2. However, since several transitions are in general possible from a given state, they exhibit a form of *nondeterminism.*

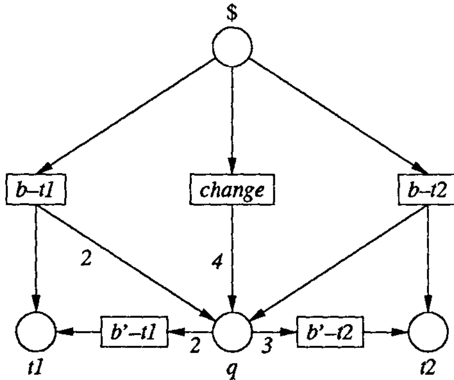
Interleaving approaches to concurrency restrict themselves to labelled transition systems or similar sequential structures. We can always sequentialize a concurrent computation (see Lemma 4) and therefore much valuable work can be and has been done in this context. However, the context as such is intrinsically sequential and forces a form of *indirect reasoning* when considering concurrency aspects; therefore, it seems quite limited. Plato's analogy of the cave[20] may provide an apt metaphor for this situation, with labelled transition systems being the wall of the cave on which the shadows of true concurrency are reflected. The metaphor seems apt because it agrees with the mathematical facts; for $\mathcal{R}$ an arbitrary rewrite theory, the descent into the cave is precisely the forgetful functor $\underline{\mathcal{R}\text{-}Sys} \longrightarrow \underline{Cat}$.

5.2 Petri Nets

This is one of the most basic models of concurrency. It has the great advantage of exhibiting concurrency *directly*, not through the indirect mediation of interleavings. Its relationship to concurrent rewriting can be expressed very simply. It is just the particular case of rewrite theories $\mathcal{N} = (\Sigma, E, L, R)$ with $\Sigma_0 = \Delta \uplus \{\lambda\}$, $\Sigma_2 = \{\otimes\}$, with all the other Σ_n empty, with $E = ACI$ —with ACI the axioms of *associativity* and *commutativity* for $\otimes$ and *identity* λ for $\otimes$— and with all terms in the rules R ground terms. Consider for example the Petri net in Figure 5, 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 rewrite theory is that of the TICKET module in the same figure.

The rules of deduction specialize as follows. The congruence rule applies just to $\otimes$ and yields instances of reflexivity for the constants. Since the rewrite rules have no variables, the replacement rule yields each of the rewrite rules as axioms. Interpreting $\otimes$ as conjunction in linear logic [8], this specialization yields sound and complete rules of deduction for the linear

[20] Republic, Bk. VII, 514-517.




```
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 5: A Petri net and its code in Maude

theory having each of the rewrite rule sequents as axioms; i.e., rewriting logic specializes in this case to *conjunctive linear logic*. The models of rewrite theories $\mathcal{N}$ of this kind are categories with a commutative monoid structure in which we have chosen certain objects —the "places"— and certain morphisms —the "transitions." The initial system $\mathcal{T}_{\mathcal{N}}$ is exactly the category $\mathcal{T}[\mathcal{N}]$ that Ugo Montanari and I associated to a Petri net as its semantics in [26, 27]. Narciso Martí-Oliet and I later studied the connection of this model with models for linear logic in [22, 21] and obtained in this way a systematic triangular correspondence between Petri nets, linear logic and categories which is a particular instance of the more general triangular correspondence between concurrent systems, rewriting logic and categories developed in this paper.

5.3 Concurrent Object-Oriented Programming

The basic syntax for objects and messages is given by the following order-sorted rewrite signature:

> *sorts Object, Attribute, Attributes, Message, Configuration, Data, Value* .
> *sorts OId, CId, AId* . `*** object, class and attribute identifiers`
> *subsorts Object, Message* < *Configuration* .
> *subsorts Attribute* < *Attributes* .
> *subsorts OId, Data* < *Value* .
> *op* $\langle _ : _ \mid _ \rangle$: *OId CId Attributes* $\longrightarrow$ *Object* .
> *op* $(_ : _)$: *AId Value* $\longrightarrow$ *Attribute* .
> *op* $_,_$: *Attributes Attributes* $\longrightarrow$ *Attributes* [*assoc comm id: nil*] .
> *op* $_\,_$: *Configuration Configuration* $\longrightarrow$ *Configuration* [*assoc comm id:* λ] .

where the operators $_\,_$ and $_,_$ are both associative and commutative with respective identities λ and *nil*. With this syntax, an *object* is represented as a term $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$ where O is the object's name, 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*. The *configuration* is the distributed state of the concurrent object-oriented system and is represented as a multiset of objects and messages. The system evolves by concurrent rewriting (modulo ACI) of the configuration by means of rewrite rules specific to each particular system, whose lefthand and righthand sides 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:

$$credit(B, M)\ \langle B : Accnt \mid bal : N \rangle \longrightarrow \langle B : Accnt \mid bal : N + M \rangle$$
$$debit(B, M)\ \langle B : Accnt \mid bal : N \rangle \longrightarrow \langle B : Accnt \mid bal : N - M \rangle.$$

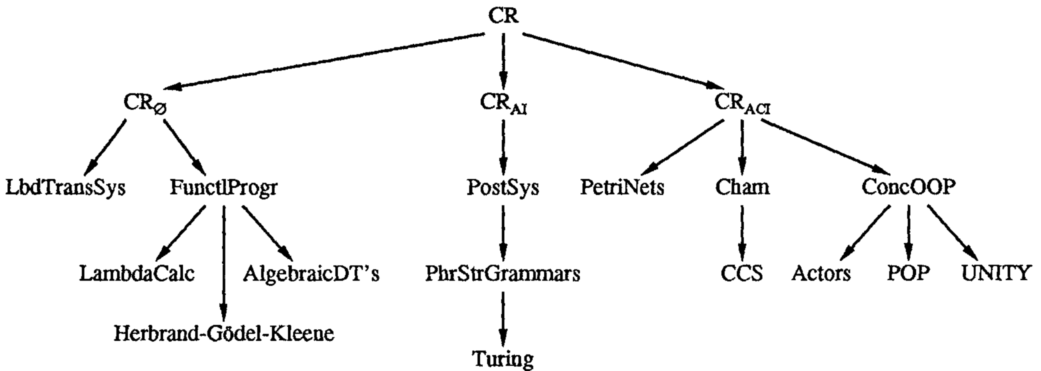


Figure 6: The Big Picture

Concurrent object-oriented systems can be defined in Maude by means of *object-oriented module definitions* of the form omod $\mathcal{O}$ endom which provide special syntax taking advantage of the structural properties common to all such systems. However, *the semantics of object-oriented modules is entirely reducible to that of system modules*, i.e., we can systematically translate an object-oriented module omod $\mathcal{O}$ endom into a corresponding system module mod $\mathcal{O}^b$ endm whose $\mathcal{O}^b$-machine semantics is the object-oriented module's intended semantics. Maude's object-oriented modules are discussed in detail in [24]; such modules share some similarities with those of FOOPS [11], and the idea of transforming objects by rewrite rules goes back to [9]. However, in comparison with FOOPS, both the treatment of concurrency and the semantics are new.

5.4 The Big Picture

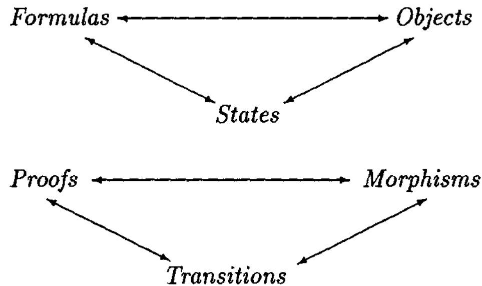
Space limitations preclude a detailed discussion of other models of concurrency to which concurrent rewriting specializes (see [25].) However, we can summarize such specializations using Figure 6, where CR stands for concurrent rewriting, the arrows indicate specializations, and the subscripts $\emptyset$, AI, and ACI stand for syntactic rewriting, rewriting modulo associativity and identity, and ACI rewriting respectively. *Functional programming* (in particular Maude's functional modules) corresponds to the case of *confluent*[21] 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. Rewriting modulo ACI includes Berry and Boudol's *chemical abstract machine* [3] (which itself specializes to CCS [28]), as well as actors [1] and Unity's model of computation [4] which can both be regarded as special cases of concurrent object-oriented programming with rewrite rules; a third special case is Engelfriet et al.'s POPs and POTs higher level Petri nets [6, 7].

6 Concluding Remarks

Within the space constraints of this paper it is impossible to do justice to the wealth of related literature on term rewriting, abstract data types, concurrency, Petri nets, linear and equational logic, ordered, continuous and nondeterministic algebras, etc. The paper [25] contains 85 such references. I would however like to mention Huet's lecture notes [16], which contains a brief discussion of rules for rewriting logic, and also work on applications of 2-categories to rewriting and to domain-theoretic and categorical approximations, including work by Rydeheard and Stell [30] and Pitts [29], whose relationship to this work is studied in [25].

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

I conclude pointing out that the model theory of rewriting logic presented here —besides yielding the general notion of concurrent system that we were seeking and providing the semantic basis for the integration of the concurrent, functional and object-oriented computational paradigms— does also establish a general *triangular correspondence* between logic, categories and concurrent systems that can be summarized as follows:



This generalizes to arbitrary rewrite systems the triangular correspondence between linear logic, Petri nets and linear categories previously developed in joint work with Narciso Martí-Oliet [22]. In particular, the correspondence between logic and categories is a Lambek-Lawvere correspondence [18, 19], a type of correspondence more abstract and general than the Curry-Howard isomorphism.

References

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

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

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

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

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

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

[7] 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.

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

[9] J.A. Goguen and J. 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Joseph A. 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.

[16] G. Huet. *Formal Structures for Computation and Deduction.* INRIA, 1986.

[17] 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.

[18] Joachim Lambek. Deductive systems and categories II. In *Category Theory, Homology Theory and their Applications I.* Springer Lecture Notes in Mathematics No. 86, 1969.

[19] F.W. Lawvere. Adjointness in foundations. *Dialectica*, 23(3/4):281–296, 1969.

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

[21] 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.

[22] 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.*

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

[24] José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990.* ACM, 1990.

[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 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.*

[27] 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.

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

[29] A. Pitts. An elementary calculus of approximations. Unpublished manuscript, University of Sussex, December 1987.

[30] D.E. Rydeheard and J.G. Stell. Foundations of equational deduction: A categorical treatment of equational proofs and unification algorithms. In *Proceedings of the Summer Conference on Category Theory and Computer Science, Edinburgh, Sept. 1987.* Springer LNCS 283, 1987.