Research Directions in Rewriting Logic*

José Meseguer

Computer Science Laboratory, SRI International
333 Ravenswood Ave. Menlo Park, CA 94025, USA

Abstract. Rewriting logic expresses an essential equivalence between logic and computation. System *states* are in bijective correspondence with *formulas*, and concurrent *computations* are in bijective correspondence with *proofs*. Given this equivalence between computation and logic, a rewriting logic axiom of the form $t \longrightarrow t'$ has two readings. Computationally, it means that a fragment of a system's state that is an instance of the pattern t can *change* to the corresponding instance of t' concurrently with any other state changes; logically, it just means that we can derive the formula t' from the formula t. Rewriting logic is entirely neutral about the structure and properties of the formulas/states t. They are entirely *user-definable* as an algebraic data type satisfying certain equational axioms. Because of this ecumenical neutrality, rewriting logic has, from a logical viewpoint, good properties as a *logical framework*, in which many other logics can be naturally represented. And, computationally, it has also good properties as a *semantic framework*, in which many different system styles and models of concurrent computation and many different languages can be naturally expressed without any distorting encodings. The goal of this paper is to provide a relatively gentle introduction to rewriting logic, and to paint in broad strokes the main research directions that, since its introduction in 1990, have been pursued by a growing number of researchers in Europe, the US, and Japan. Key theoretical developments, as well as the main current applications of rewriting logic as a logical and semantic framework, and the work on formal reasoning to prove properties of specifications are surveyed.

Table of Contents

1 Introduction

2 An Introduction to Rewriting Logic
- 2.1 Inference Rules and their Meaning
- 2.2 A Computational-Logical Example: Petri Nets
- 2.3 Models
- 2.4 2-Category Models
- 2.5 Reflection and Strategies

* Supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, and by National Science Foundation Grant CCR-9633363.

3 A Semantic Framework for Concurrency Models
3.1 Parallel Functional Programming
3.2 Labeled Transition Systems
3.3 Grammars
3.4 Petri Nets and Algebraic Nets
3.5 Gamma and the Chemical Abstract Machine
3.6 CCS, LOTOS and the π-Calculus
3.7 Concurrent Objects, Actors, and OO Databases
3.8 Unity
3.9 Graph Rewriting
3.10 Dataflow
3.11 Neural Networks
3.12 Real-Time Systems
3.13 Tile Models

4 A Semantic Framework for Programming Languages
4.1 Rewriting Logic Languages
Executable Specification Languages.
Concurrent Rewriting Logic Languages.
4.2 Executable Programming Language Specification

5 A Semantic Framework for Software Architectures
5.1 From Object-Oriented Design to Rewriting Logic
5.2 Rewriting Semantics of Software Components

6 Logical Framework Applications
6.1 Representing Logics
6.2 Specifying and Building Formal Methods Tools
6.3 Metaprogramming
6.4 Avoiding the Frame Problem
6.5 Towards Formal Interoperability

7 Formal Reasoning, Refinement, and Transformation
7.1 Refinement and Transformation
7.2 Modal and Temporal Logics

8 Concluding Remarks

1 Introduction

Rewriting logic [113, 115] is like a two-faced Janus, in that it expresses an essential equivalence between logic and computation in a particularly simple way. Namely, system *states* are in bijective correspondence with *formulas* (modulo whatever structural axioms are satisfied by such formulas: for example, modulo the associativity and commutativity of a certain connective) and concurrent *computations* in a system are in bijective correspondence with *proofs* (modulo appropriate notions of equivalence between computations and between proofs).

Given this equivalence between computation and logic, a rewriting logic axiom of the form

$$t \longrightarrow t'$$

has two readings. Computationally, it means that a fragment of a system's state that is an instance of the pattern t can *change* to the corresponding instance of t' concurrently with any other state changes; that is, the computational reading is that of a *local concurrent transition*. Logically, it just means that we can derive the formula t' from the formula t; that is, the logical reading is that of an *inference rule*.

Rewriting logic is entirely neutral about the structure and properties of the formulas/states t. They are entirely *user-definable* as an algebraic data type satisfying certain equational axioms, so that rewriting deduction takes place *modulo* such axioms. Because of this ecumenical neutrality, rewriting logic has, from a logical viewpoint, good properties as a *logical framework* [107], in which many other logics can be naturally represented. And, computationally, it has also good properties as a *semantic framework* [118], in which many different system styles and models of concurrent computation and many different languages can be naturally expressed without any distorting encodings.

In the spirit of the Marktoberdorf lectures—but with extensive pointers to the literature and incorporating some more recent advances—the goal of this paper is to provide a relatively gentle introduction to rewriting logic (Section 2) and to paint in broad strokes the main research directions that, since its introduction in 1990, have been pursued by a growing number of researchers in Europe, the US and Japan. A task of this nature necessarily involves one's own personal views and preferences, but I have seriously tried to discuss all the research contributions that I am aware of. Besides key theoretical developments, I survey the main current applications of rewriting logic as a logical and semantic framework, and the work on formal reasoning about rewriting logic specifications.

As a semantic framework, rewriting logic supports a wide spectrum of applications. Firstly, from its inception it has been used to specify and unify many different *models of concurrent computation*. How many such models can be very naturally expressed within rewriting logic is the subject of Section 3, that summarizes and updates [118]. Secondly, rewriting logic is also a semantic framework for *programming languages* in a twofold sense: it can itself be used as an executable specification language and as a concurrent declarative language; and it can also be used to formally define and prototype many other languages, including concurrent and distributed ones. Section 4 describes the language applications in these two senses, including the several rewriting logic languages that have been developed in Europe, the US and Japan. Thirdly, *software architectures* describing the high-level structure of software systems and their composition out of software *components* is an area very much in need of formal foundations, where rewriting logic is beginning to play an important role (Section 5).

As a logical framework, the applications of rewriting logic seem very promising. Firstly, one can represent other logics in it, and, using an implementation of rewriting logic, one can use those representations to execute the logics so

represented. Secondly, the same idea can be applied not just to a logic, but to the *inference system* of a theorem prover or some other automated deduction tool; in this way, we can formally specify and execute the theorem prover or tool in rewriting logic, and we can even use the specification as the actual tool. All these representation maps are instances of a very powerful *metaprogramming* capability, that is available in rewriting logic thanks to its reflective features. In summary, perhaps the main goal that rewriting logic can help us achieve as a logical framework is *logical interoperability*, that is, the capacity to move in a mathematically rigorous way across different formal systems and to correctly interoperate their associated formal tools.

Section 7 discusses work on formal reasoning about rewriting logic specifications. This reasoning can happen within rewriting logic itself—involving inductive reasoning about its initial models, and different kinds of formal analysis, refinement and transformation—or can happen in a more abstract, property-oriented logic, such as a modal or temporal logic, used to prove global properties of the system that we have specified. In either case, rewriting logic can be used as a logical framework to combine and interoperate the different formal methods tools supporting our reasoning. The paper ends with some remarks about future directions in Section 8.

2 An Introduction to Rewriting Logic

The inference rules and their meaning is first explained. This meaning is further illustrated with a Petri net example. Models, and their 2-category functorial semantics are also discussed. Reflection and strategies are introduced.

2.1 Inference Rules and their Meaning

A *signature* in rewriting logic is an equational theory[2] (Σ, E), where Σ is an equational signature and E is a set of Σ-equations. Rewriting will operate on equivalence classes of terms modulo 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 of equations E is empty. Techniques for rewriting modulo equations have been studied extensively [46] and can be used to implement rewriting modulo many equational theories of interest.

Given a signature (Σ, E), *sentences* of rewriting logic are sequents of the form

$$[t]_E \longrightarrow [t']_E,$$

[2] Rewriting logic is parameterized by the choice of its underlying equational logic, that can be unsorted, many-sorted, order-sorted, membership equational logic, and so on. To ease the exposition I give an *unsorted* presentation.

where t and t' are Σ-terms possibly involving some variables, and $[t]_E$ denotes the equivalence class of the term t modulo the equations E. A *rewrite theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of *labels*, and R is a set of pairs $R \subseteq L \times T_{\Sigma,E}(X)^2$ whose first component is a label and whose second component is a pair of E-equivalence classes of terms, with $X = \{x_1, \ldots, x_n, \ldots\}$ a countably infinite set of variables. Elements of R are called *rewrite rules.*[3] We understand a rule $(r, ([t], [t']))$ as a labeled sequent and use for it the notation $r : [t] \longrightarrow [t']$. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either t or t', we write $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, or in abbreviated notation $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$.

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sentence $[t] \longrightarrow [t']$, or that $[t] \longrightarrow [t']$ is a *(concurrent) $\mathcal{R}$-rewrite*, 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* (where we assume that all the terms are well formed and $t(\overline{w}/\overline{x})$ denotes the simultaneous substitution of w_i for x_i in t):

1. **Reflexivity**. For each $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] \longrightarrow [t]}$.
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 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**
$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}.$$

Rewriting logic is a logic for reasoning correctly about *concurrent systems* having *states*, and evolving by means of *transitions*. The signature of a rewrite theory describes a particular structure for the states of a system—e.g., multiset, binary tree, etc.—so that its states can be distributed according to such a structure. The rewrite rules in the theory describe which *elementary local transitions* are possible in the distributed state by concurrent local transformations. The rules of rewriting logic allow us to reason correctly about which *general* concurrent transitions are possible in a system satisfying such a description. Thus,

[3] To simplify the exposition the rules of the logic are given for the case of *unconditional* rewrite rules. However, all the ideas presented here have been extended to conditional rules in [115] 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 increases considerably the expressive power of rewrite theories.

computationally, each rewriting step is a parallel local transition in a concurrent system.

Alternatively, however, we can adopt a logical viewpoint instead, and regard the rules of rewriting logic as *metarules* for correct deduction in a *logical system*. Logically, each rewriting step is a logical *entailment* in a formal system.

The computational and the logical viewpoints under which rewriting logic can be interpreted can be summarized in the following diagram of correspondences:

State		$\leftrightarrow$ *Term*		$\leftrightarrow$ *Proposition*	
Transition		$\leftrightarrow$ *Rewriting*		$\leftrightarrow$ *Deduction*	
Distributed	*Structure*	$\leftrightarrow$ *Algebraic*	*Structure*	$\leftrightarrow$ *Propositional*	*Structure*

The last row of equivalences is actually quite important. Roughly speaking, it expresses the fact that a state can be transformed in a concurrent way only if it is nonatomic, that is, if it is *composed* out of smaller state components that can be changed independently. In rewriting logic this composition of a concurrent state is formalized by the *operations* of the signature Σ of the rewrite theory $\mathcal{R}$ that axiomatizes the system. From the logical point of view such operations can naturally be regarded as user-definable *propositional connectives* stating the particular structure that a given state has. A subtle additional point about the last row of equivalences is that the algebraic structure of a system also involves *equations*. Such equations describe the system's global state as a *concurrent data structure*; they can have a dramatic impact on the amount of concurrency available in a system.

Note that it follows from this discussion that rewriting logic is primarily a logic *of* change—in which the deduction directly corresponds to the change—as opposed to a logic to talk *about* change in a more indirect and global manner such as the different variants of modal and temporal logic. As further discussed in Section 7.2, these latter logics support a nonexecutable—as far as the system described is concerned—level of specification above that of rewriting logic.

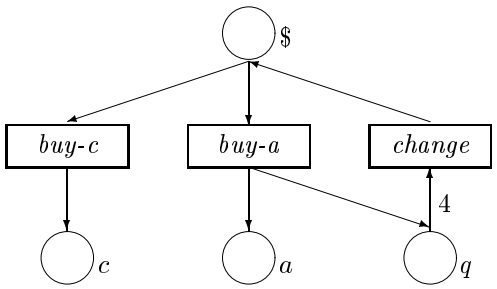
2.2 A Computational-Logical Example: Petri Nets

Petri nets [144] provide a simple, yet rich enough, example of systems exhibiting concurrent change. Therefore, the problem of representing in logical terms the concurrent change of a Petri net system provides an interesting first illustration of how rewriting logic formalizes such change. In addition, since the state of a Petri net can be viewed as a conjunction of elementary resources, Petri net reachability provides a simple example of conjunctive planning. The discussion here is adapted from [108], where the Petri net case is related to other logical approaches to planning.

Usually, a Petri net is presented as a set of places, a disjoint set of transitions and a relation of causality between them that associates to each transition the set of resources consumed as well as produced by its firing. Meseguer and Montanari recast this idea in an algebraic framework in [122]. From this point of view, resources are represented as multisets of places, and therefore we have a binary

operation (multiset union, denoted $\otimes$ here) that is associative, commutative, and has the empty multiset as an identity[4] but is not idempotent. Then, a Petri net is viewed as a graph whose arcs are the transitions and whose nodes are multisets over the set of places, usually called *markings*.

The following Petri net represents a machine to buy cakes and apples; a cake costs a dollar and an apple three quarters. Due to an unfortunate design, the machine only accepts dollars, and it returns a quarter when the user buys an apple; to alleviate in part this problem, the machine can change four quarters into a dollar.



As a graph, this net has the following arcs:

$$\begin{array}{ll} \textit{buy-}c: & \$ \longrightarrow c \\ \textit{buy-}a: & \$ \longrightarrow a \otimes q \\ \textit{change}: & q \otimes q \otimes q \otimes q \longrightarrow \$ \end{array}$$

The expression of this Petri net in rewriting logic is now obvious. We can view each of the labeled arcs of the Petri net as a rewrite rule in a rewrite theory having a binary *associative, commutative* operator $\otimes$ (multiset union) with *identity* `1` so that rewriting happens modulo ACI, that is, is multiset rewriting. Adopting an order-sorted [65] version of rewriting logic, we can gather together the "places" $\$, q, a, c$ into a type `Place` and view the states of the net, usually called *markings*, as elements of a supertype `Marking` containing `Place` and endowed with a multiset union operator. Using the syntax of the Maude language, the rewrite theory corresponding to the above Petri net can then be expressed as follows:

```
mod PETRI-NET is
  sorts Place Marking .
  subsort Place < Marking .
  op 1 : -> Marking .
  op _⊗_ : Marking Marking -> Marking [assoc comm id: 1] .
  ops $ q a c : -> Place .
```

[4] From now on the associativity, commutativity, and identity axioms are denoted by the acronym ACI.

```
  rl buy-c : $ => c .
  rl buy-a : $ => a ⊗ q .
  rl change : q ⊗ q ⊗ q ⊗ q => $ .
endm
```

A Maude module contains sort and subsort declarations introduced by the keywords `sort(s)` and `subsort(s)` stating the different sorts of data manipulated by the module and how those sorts are related. Each of the operators declared in 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, infix ($_ \otimes _$, for example) or any "mixfix" combination as well as standard parenthesized notation.

The sense in which rewriting logic provides a sound and complete logical calculus to answer questions about what states can be reached in a Petri net is made precise by the following

Theorem 1. *[115] Let N be a Petri net with set of places S and set of transitions T, let $N^\sharp$ be the rewrite theory representing such a net as explained above, and let M, M' be markings on S. Then, the marking M' is reachable from M in the net N if and only if the rewrite $[M] \longrightarrow [M']$ is a provable consequence of the theory $N^\sharp$ using the rules of deduction of rewriting logic.*

In rewriting logic, terms have a dual computational-logical interpretation. They can be interpreted as system *states*, or they can be interpreted as logical *formulas*; similarly, rewrites can be interpreted as system *transitions* or as logical *inferences*. Which viewpoint should be favored may depend on the particular interests. For example, the multiset union operator $\otimes$ can be viewed as a form or resource-conscious non-idempotent conjunction. Then, the state `a ⊗ q ⊗ q` corresponds to having an apple *and* a quarter *and* a quarter, which is a strictly better situation than having an apple *and* a quarter (non-idempotence of $\otimes$). Several researchers realized independently that this ACI operation on multisets corresponds to the conjunctive connective $\otimes$ (*tensor*) in linear logic [7, 161, 68, 103, 104]. This complementary point of view sees a net as a theory in this fragment of linear logic. Then, it is possible to establish a precise connection between reachability in Petri nets and provability in tensor logic.

For example, in order to get the tensor theory corresponding to our Petri net above, it is enough to change the arrows in the graph presentation into turnstiles, getting the following axioms:

$$\begin{array}{ll} \textit{buy-}c: & \$ \vdash c \\ \textit{buy-}a: & \$ \vdash a \otimes q \\ \textit{change}: & q \otimes q \otimes q \otimes q \vdash \$ \end{array}$$

Note that there are no variables involved in the rules (where the rules now are viewed as axioms in tensor logic), i.e., all the terms involved are ground. Therefore, in this case, the *Replacement* rule of rewriting logic does not require any arguments, and the rules of rewriting logic reduce to the rules of tensor logic,

the fragment of propositional linear logic consisting only of the connective $\otimes$ and its neutral element 1 (see [115, Sections 5.3.1–2]). Therefore, the following triple equivalence (including the previous equivalence in Theorem 1) is immediate:

Theorem 2. *[103, 115] Let N be a Petri net with set of places S and set of transitions T, and M, M' be markings on S. Then, the marking M' is reachable from M if and only if the sequent $M \vdash M'$ is provable in tensor logic from the axioms corresponding to T, if and only if there is an $N^\sharp$-rewrite $[M] \longrightarrow [M']$ (where $N^\sharp$ denotes the corresponding rewrite theory).*

Of course, since the symbols in the signature Σ of a rewrite theory used to build formulas are entirely user-definable, and they can satisfy structural properties axiomatized by whatever equational axioms E one chooses, rewriting logic can represent formulas in many logics, expressing the inference rules as rewrite rules. In this sense, linear logic does not play any privileged role; it is just one of the many logics that can be thus faithfully represented.

2.3 Models

This section could be skipped in a first reading—or could be postponed until the rest of the paper has been read—without much loss to the main flow of the ideas.

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. By adopting a logical instead of a computational perspective, we can alternatively view such models as "logical systems" in which formulas are validly rewritten to other formulas by concurrent rewritings which correspond to proofs for the logic in question. Such models have a natural *category* structure, with states (or formulas) as objects, transitions (or proofs) as morphisms, and sequential composition as morphism composition, and in them dynamic behavior exactly corresponds to deduction.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, for which we assume that different labels in L name different rules in 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 domains and codomains, are given below; they just "decorate" with proof terms the rules 1–4 of rewriting logic. Note that we always use "diagrammatic" notation for morphism composition, i.e., $\alpha; \beta$ always means the composition of α *followed by* β.

1. **Identities**. For each $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] : [t] \longrightarrow [t]}$.
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** $\dfrac{\alpha : [t_1] \longrightarrow [t_2] \qquad \beta : [t_2] \longrightarrow [t_3]}{\alpha; \beta : [t_1] \longrightarrow [t_3]}$.

Each of the above rules of generation defines a different operation taking certain proof terms as arguments and returning a resulting proof term. In other words, proof terms form an algebraic structure $\mathcal{P}_{\mathcal{R}}(X)$ consisting of a graph with nodes $T_{\Sigma,E}(X)$, with identity arrows, and with operations f (for each $f \in \Sigma$), r (for each rewrite rule), and $_;_$ (for composing arrows). Our desired model $\mathcal{T}_{\mathcal{R}}(X)$ is the quotient of $\mathcal{P}_{\mathcal{R}}(X)$ modulo the following equations:[5]

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 $f \in \Sigma_n$,
   (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 $\alpha_1, \ldots, \alpha_n$, $\quad t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n)$.
4. **Exchange**. For each $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,

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

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

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

Since each proof term is a description of a concurrent computation, what these equations provide is an equational theory of *true concurrency* allowing us to characterize when two such descriptions specify the same abstract computation. We shall see in Section 3.4 that for Petri nets this notion of true concurrency coincides with the well-known notion of commutative processes of a net.

Note that, since $[t(x_1,\ldots,x_n)]$ and $[t'(x_1,\ldots,x_n)]$ can both be regarded as functors $\mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$, from the mathematical point of view the exchange law just asserts that r is a *natural transformation*.

Lemma 3. *[115] For each rewrite rule* $r : [t(x_1,\ldots,x_n)] \longrightarrow [t'(x_1,\ldots,x_n)]$ *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 $r : [t(x_1,\ldots,x_n)] \Rightarrow [t'(x_1,\ldots,x_n)]$ *between the functors* $[t(x_1,\ldots,x_n)], [t'(x_1,\ldots,x_n)] : \mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$.

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

Definition 4. Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, an $\mathcal{R}$*-system* $\mathcal{S}$ is a category $\mathcal{S}$ together with:

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

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

  satisfying the equations E, i.e., for any $t(x_1,\ldots,x_n) = t'(x_1,\ldots,x_n)$ in E we have an identity of functors $t_{\mathcal{S}} = t'_{\mathcal{S}}$, where the functor $t_{\mathcal{S}}$ is defined inductively from the functors $f_{\mathcal{S}}$ in the obvious way.
- for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R a natural transformation $r_{\mathcal{S}} : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}$.

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[6] $r_{\mathcal{S}} * F = F^n * r_{\mathcal{S}'}$, where n is the number of variables appearing in the rule. This defines a category $\mathcal{R}$-**Sys** in the obvious way.

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 [115], where the soundness and completeness of rewriting logic for $\mathcal{R}$-system models is also proved.

Theorem 5. $\mathcal{T}_{\mathcal{R}}$ *is an initial object in the category* $\mathcal{R}$-**Sys**. *More generally,* $\mathcal{T}_{\mathcal{R}}(X)$ *has the following universal property: Given an* $\mathcal{R}$*-system* $\mathcal{S}$*, each function* $F : X \longrightarrow |\mathcal{S}|$ *extends uniquely to an* $\mathcal{R}$*-homomorphism* $F^\natural : \mathcal{T}_{\mathcal{R}}(X) \longrightarrow \mathcal{S}$.

[6] Note that we use diagrammatic order for the *horizontal*, $\alpha * \beta$, and *vertical*, $\gamma;\delta$, composition of natural transformations [102].

2.4 2-Category Models

This section can be skipped in a first reading; it provides useful background for the discussion on tile models in Section 3.13.

Lawvere [89] made the seminal discovery that, given an equational theory $T = (\Sigma, E)$ and a Σ-algebra A satisfying E, the assignment to each E-equivalence class $[t(x_1, \ldots, x_n)]$ of its associated functional interpretation in A, $A_{[t]} : A^n \longrightarrow A$ is in fact a product-preserving functor $\hat{A} : \mathcal{L}_T \longrightarrow \mathbf{Set}$, where the category $\mathcal{L}_T$, called T's "Lawvere theory," has objects the natural numbers, with E-equivalence classes of terms $[t(x_1, \ldots, x_n)]$ as arrows $[t(x_1, \ldots, x_n)] : n \longrightarrow 1$, and with composition of arrows given by substitution. That is,

$$m \xrightarrow{([u_1], \ldots, [u_n])} n \xrightarrow{[t]} 1$$

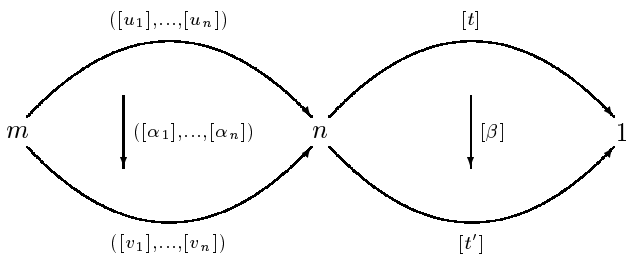
yields as a composition $[t(\overline{u}/\overline{x})] : m \longrightarrow 1$. The product structure of $\mathcal{L}_T$ is very simple, namely, n is the n-fold product of 1 with itself, and has projections $[x_1], \ldots, [x_n]$. Choosing canonical set-theoretic products in the targets of such functors, and denoting by $\mathbf{Mod}(\mathcal{L}_T, \mathbf{Set})$ the category with objects those functors, and with morphisms natural transformations between them, the assignment $A \mapsto \hat{A}$ becomes an isomorphism of categories

$$\mathbf{Alg}_{\Sigma,E} \cong \mathbf{Mod}(\mathcal{L}_T, \mathbf{Set})$$

where $\mathbf{Alg}_{\Sigma,E}$ is the category of T-algebras.

As pointed out in [114], this situation generalizes very naturally to the case of rewriting logic, where models are algebraic structures on categories instead than on sets. That is, the *ground* on which they exist is the 2-category [102, 78] **Cat**, instead of the category **Set**. Intuitively, $\mathcal{C}$ is a 2-category when the morphisms $\mathcal{C}(A, B)$ between two objects form not just a set, but a category, and the two arrow compositions fit together in a coherent way. In **Cat**, $\mathbf{Cat}(A, B)$ is the category with objects the functors from A to B, and with morphisms the natural transformations between such functors.

Given an (unsorted and unconditional) rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ we define a 2-category with 2-products $\mathcal{L}_\mathcal{R}$ where the objects are the natural numbers, the category $\mathcal{L}_\mathcal{R}(n, 1)$ has as objects E-equivalence classes of terms $[t(x_1, \ldots, x_n)]$, and as morphisms equivalence classes of proof terms $[\alpha] : [t(x_1, \ldots, x_n)] \longrightarrow t'(x_1, \ldots, x_n)]$ with (vertical) composition given by $[\alpha]; [\beta] = [\alpha; \beta]$. The horizontal composition of proofs



is then given by the proof term $[t(\overline{\alpha}/\overline{x}); \beta(\overline{v}/\overline{x})] : [t(\overline{u}/\overline{x})] \longrightarrow [t'(\overline{v}/\overline{x})]$.

The point is that, as mentioned in [114], $\mathcal{L}_{\mathcal{R}}$ does for $\mathcal{R}$-systems what in the **Set** case $\mathcal{L}_T$ does for T-algebras. That is, given an $\mathcal{R}$-system $\mathcal{S}$ the assignment to each rule $r : [t] \longrightarrow [t']$ in R of a natural transformation $r_{\mathcal{S}} : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}$ between the functors $t_{\mathcal{S}}, t'_{\mathcal{S}} : \mathcal{S}^n \longrightarrow \mathcal{S}$ extends naturally to a 2-product preserving 2-functor $\hat{\mathcal{S}} : \mathcal{L}_{\mathcal{R}} \longrightarrow \mathbf{Cat}$, and the assignment $\mathcal{S} \mapsto \hat{\mathcal{S}}$ yields an isomorphism of 2-categories

$$\mathcal{R}\text{-}\mathbf{Sys} \cong \mathbf{Mod}(\mathcal{L}_{\mathcal{R}}, \mathbf{Cat})$$

where $\mathbf{Mod}(\mathcal{L}_{\mathcal{R}}, \mathbf{Cat})$ is defined as the category of canonical 2-product preserving 2-functors from $\mathcal{L}_{\mathcal{R}}$ to **Cat**.

Therefore, models of rewriting logic have a natural 2-categorical interpretation in which $\mathcal{L}_{\mathcal{R}}$ plays the role of a generic model among all the 2-category models of $\mathcal{R}$. This point of view has been further generalized and exploited in Pisa to provide very useful connections with other concurrency theory models. Corradini, Gadducci and Montanari [37] provide a uniform construction for $\mathcal{L}_{\mathcal{R}}$ and for a *sesqui-category* model, similar to $\mathcal{L}_{\mathcal{R}}$ but satisfying fewer equations, that has been proposed by Stell [151]. They associate posets of partial orders of events to both models, and make the important observation that when a rewrite rule is not right linear—that is, when it has a repeated occurrence of a variable in its righthand side—then the poset associated to $\mathcal{L}_{\mathcal{R}}$ is not a prime algebraic domain, whereas the poset of the sesqui-category model is. In this way, the relationship between rewriting logic models and event structures is clarified. What happens is that, when rules are not right linear, $\mathcal{L}_{\mathcal{R}}$ is in a sense too abstract, because what is one event in one proof term may— because of repetition of variables—become several events in a proof term equivalent to it by the exchange axiom; in the sesqui-category model the exchange axiom does not hold, and therefore those computations are considered different.

2.5 Reflection and Strategies

Intuitively, a logic is reflective if it can represent its metalevel at the object level in a sound and coherent way. Reflection is a very useful property in computing systems and therefore very desirable in a computational logic. Manuel Clavel and I [33] have given general axioms centered around the notion of a *universal theory* that a logic should satisfy to properly be called reflective. We have also shown that rewriting logic is reflective [33, 34, 29] in the precise sense that there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that for any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) we have the following equivalence

$$\mathcal{R} \vdash t \rightarrow t' \;\; \Leftrightarrow \;\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle,$$

where $\overline{\mathcal{R}}$ and $\overline{t}$ are terms representing $\mathcal{R}$ and t as data elements of $\mathcal{U}$. Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

$$\mathcal{R} \vdash t \rightarrow t' \;\Leftrightarrow\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \;\Leftrightarrow\; \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \rightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots$$

Reflection is a very powerful property. It is systematically exploited in the Maude rewriting logic language implementation [31], that provides key features of the universal theory $\mathcal{U}$ in a built-in module called `META-LEVEL`. In particular, `META-LEVEL` has sorts `Term` and `Module`, so that the representations $\overline{t}$ and $\overline{\mathcal{R}}$ of a term t and a module $\mathcal{R}$ have sorts `Term` and `Module`, respectively. `META-LEVEL` has also functions `meta-reduce`$(\overline{\mathcal{R}}, \overline{t})$ and `meta-apply`$(\overline{\mathcal{R}}, \overline{t}, \overline{l}, \overline{\sigma}, n)$ which return, respectively, the representation of the reduced form of a term t using the equations in a module $\mathcal{R}$, and the (representation of the) result of applying a rule labeled l in the module $\mathcal{R}$ to a term t at the top with the $(n+1)$th match consistent with the partial substitution σ. As the universal theory $\mathcal{U}$ that it implements in a built-in fashion, `META-LEVEL` can also support a reflective tower with an arbitrary number of levels of reflection.

Reflection can be exploited in many useful ways. One important area of application is *internal rewriting strategies* [34, 35, 29], that is, strategies to guide the rewriting process whose semantics can be defined inside the logic by rewrite rules at the metalevel. Since rewrite theories need not be Church-Rosser and may not terminate, they can in general be executed in many different ways, yielding widely different behaviors. Therefore, strategies guiding the application of the rules are very important.

A reflective rewriting logic language like Maude allows defining such strategies by rewrite rules inside the language. It even allows a reflective tower of strategies, with meta-strategies controlling the application of rules for strategies, and so on. The basic idea is that `META-LEVEL` can be used as a *reflective kernel* that can be extended in a completely user-definable way to specify *internal strategy languages* controlling the rewriting process. Intuitively, `meta-apply` allows us to take *elementary steps* in the deduction process, but we can take *bigger steps* by defining more complex *strategy expressions* in a module extending `META-LEVEL` by *semantic equations* and *semantic rules*, that define in rewriting logic the precise rewriting semantics for each of the constructors building up strategy expressions in our strategy language of choice. Several simple, yet powerful, strategy languages of this kind are discussed in [32, 35, 29, 31]. The idea of expressing strategies with rewrite rules and some reflective properties have also been adopted by more recent work on ELAN [17, 82]. The ELAN system [79, 20] has been the first rewriting logic language providing good user support for strategies, in what might now be called an external strategy language (see [18, 19] for the most recent work on ELAN strategies).

Rewriting logic reflection has many other uses; I discuss some of them in Section 6. An important and very general use—intimately connected with the logical and semantic framework applications discussed in this paper—is defining and executing *within* rewriting logic itself *representation maps* of the form

$$\Phi : \mathcal{L} \longrightarrow \mathit{RWLogic}.$$

Such maps give a rewriting logic semantics to a language $\mathcal{L}$ by associating to each module M in $\mathcal{L}$ a rewrite theory $\Phi(M)$ in *RWLogic*. In semantic framework applications the language $\mathcal{L}$ can be a model of computation, a programming

language, or an architectural description language; in logical framework applications $\mathcal{L}$ is typically a logic. In all cases, both $\mathcal{L}$ and Φ are metalevel entities that are, in principle, outside rewriting logic; however, thanks to reflection they can be *internalized*—or as it is sometimes said *reified*—within rewriting logic. The idea is that in the universal theory $\mathcal{U}$ rewrite theories are already reified in an algebraic data type `Module`; we can then define another such data type $\texttt{Module}_{\mathcal{L}}$ reifying the modules in $\mathcal{L}$, and can reify Φ as a function

$$\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \longrightarrow \texttt{Module}.$$

Since typically Φ is a total computable function, by general results of Bergstra and Tucker [11] it can always be specified by Church-Rosser and terminating rewrite rules, and therefore can be defined and executed in a reflective rewriting logic language such as Maude.

3 A Semantic Framework for Concurrency Models

Regarding the computational uses of rewriting logic, an obvious question to ask is how general and natural rewriting logic is as a *semantic framework* in which to express different languages and models of computation. This section presents concrete evidence for the thesis that a wide variety of models of computation, including concurrent ones, can be naturally and directly expressed as rewrite theories in rewriting logic without any encoding or artificiality. As a consequence, models hitherto quite distant from each other can be naturally unified and interrelated within a common framework.

3.1 Parallel Functional Programming

Functional computations, although amenable to parallelization, are nevertheless *determinate*, in the sense that the final result of a functional expression is the unique value—if it exists—computed by the composition of functions described in the expression. The Church-Rosser property is the technical property guaranteeing such determinacy. We can distinguish between *first-order* functional languages, also called *equational* languages, in which programs are collections of functions defined by Church-Rosser equational theories, and *higher-order* languages that are typically based on some typed or untyped lambda calculus, so that functions can be defined by lambda expressions, where the lambda calculus in question also satisfies the Church-Rosser property.

Of course, first-order functional programming can be viewed as the special case in which the rewrite rules *are* Church-Rosser. In this way, a seamless integration of parallel functional programming within the more general framework of rewriting logic is naturally achieved. More abstractly, such an integration can be viewed as a conservative embedding of equational logic within rewriting logic [105]. For higher-order functions, the key observation is that rewriting logic allows rewriting *modulo* equational axioms. We can then take advantage of the different reductions of lambda calculi to first-order equational logic using an

equational theory of *explicit substitution* to view lambda calculus reduction as first-order rewriting *modulo* the substitution equations. In fact, in several formalizations congruence modulo substitution exactly corresponds to alpha-conversion equivalence between lambda terms.

The natural inclusion of the lambda calculus within rewriting logic using explicit substitution was pointed out in [115]. An illuminating investigation of parallel computations in the lambda calculus using rewriting logic has been carried out by Laneve and Montanari [87, 88], who have considered the even more general case of *orthogonal, left-normal combinatory reduction systems* as formalized by Aczel [1], that contain the lambda calculus as a special case. They show that such systems exactly correspond to rewrite theories $\mathcal{R}$ whose equational part E consists of explicit substitution equations. They then prove that the traditional model of parallel rewriting in such systems—generalizing parallel lambda calculus rewriting—exactly corresponds to a quotient of $\mathcal{T}_{\mathcal{R}}(X)$ by a few equations. In this way, they obtain a simple and purely equational theory of equivalence or "true concurrency" between parallel lambda calculus computations that is considerably simpler than that afforded by the heavy machinery of the residual calculus.

3.2 Labeled Transition Systems

A labeled transition system is a poor man's rewrite theory. It is just a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ in which Σ consists only of constants, E is empty, and the rules are all of the form $r : a \longrightarrow b$ with a and b some constants in Σ.

A labeled transition system's poverty has two aspects. Firstly, it is very *low-level*, since the states are unstructured atomic entities so that infinite state spaces need some form of schematic presentation; also, the rules apply only to individual transitions, whereas for general rewrite theories a single rule may cover an infinite number of them. Secondly, and more importantly, *a labeled transition system can be nondeterministic, but it cannot exhibit concurrency*. The reason for this is the negative side of our motto

$$\textit{Distributed Structure} \leftrightarrow \textit{Algebraic Structure}$$

Since the states are *atomic* entities, they do not have parts that can evolve concurrently. For a system to be concurrent its states must be *decomposable*. This is what the nonconstant operators in a signature Σ make possible. Petri nets are also low level, automaton-like systems, but they are concurrent precisely because there is a binary multiset union operator composing their distributed states.

3.3 Grammars

Traditional grammars for formal languages are just string rewriting systems. They can be concurrent, because different rewrites may simultaneously transform different substrings. The most general such grammars are Post systems;

their parallelism is meant to model that of logical deductions in a formal system. Phrase-structure grammars are more restrictive, because they only involve ground terms in their rewrite rules. Turing machines, viewed as grammars, are even more restrictive. All of them can be naturally viewed as rewrite theories having a signature with $\Sigma_0 = \Delta \uplus \{\lambda\}$ (with Δ the alphabet, and λ the empty string), $\Sigma_2 = \{__\}$ (the binary string concatenation operator), and all the other Σ_n are empty. The equational axioms E are in this case the *associativity* of string concatenation and the *identity* axioms for concatenation with λ. Therefore, $T_{\Sigma,AI} = \Delta^*$, and $T_{\Sigma,AI}(X) = (\Delta \uplus X)^*$. The rules of a rewrite theory for this case must have the form:

$$u_0\, X_{k_1}\, u_1 \ldots u_{n-1}\, X_{k_n}\, u_n \longrightarrow v_0\, X_{l_1}\, v_1 \ldots v_{m-1}\, X_{l_m}\, v_m$$

with $n, m \in \mathbb{N}$, $u_i, v_j \in \Delta^*$, where the variables $X_{k_i}, X_{l_j} \in X$ could actually be repeated, i.e., we could have $X_{k_i} = X_{k_{i'}}$ with $i \neq i'$ and similarly for the X_l's.

3.4 Petri Nets and Algebraic Nets

The naturalness with which Petri nets and their concurrent computations can be expressed in rewriting logic has already been illustrated in Section 2.2. Furthermore, it is quite remarkable that when $\mathcal{N}$ is the rewrite theory associated to a Petri net, the general initial model construction $\mathcal{T}_{\mathcal{N}}$ described in Section 2.3 specializes to a well-known "true concurrency" model for nets. Namely, when the finite concurrent computations of a net $\mathcal{N}$ are formalized as arrows in the category $\mathcal{T}_{\mathcal{N}}$, they exactly correspond to the *commutative processes* of $\mathcal{N}$ in the sense of Best and Devillers [13]. This result has been proved by Degano, Meseguer and Montanari [40, 41] using an earlier categorical model of Petri net computations denoted $\mathcal{T}[\mathcal{N}]$ [122] that is in fact identical to $\mathcal{T}_{\mathcal{N}}$.

Since Petri nets are in some ways a very simple concurrency model, in practice it is often convenient to specify systems at a higher level, yet using the same basic properties of Petri nets. That is, instead of atomic places one wants to have *structured data*, perhaps equationally axiomatized by algebraic data types. This is the analogue for Petri nets of what languages like LOTOS provide for process algebras, since in both cases the practical need to support data types is very similar. Rewriting logic offers a very natural framework for giving semantics to different kinds of *algebraic* Petri nets of this kind. For the case of Engelfriet et al.'s higher level Petri nets, called POPs [52, 53], this was pointed out in [115]. Applications of rewriting logic to Petri net algebraic specification have been developed by Battiston, Crespi, De Cindio, and Mauri [9], and also by Bettaz and Maouche [14, 15]. More recently, stimulated by the lectures at Marktoberdorf, Mark-Oliver Stehr has defined a systematic translation from a very general format of algebraic Petri nets to rewriting logic, has proposed suitable equational axioms for the resulting rewrite theories, and has demonstrated the use of Maude as an execution environment for algebraic Petri nets through several case studies [150].

3.5 Gamma and the Chemical Abstract Machine

The Gamma language of Banâtre and Le Mètayer [8], and Berry and Boudol's *chemical abstract machine*, or *cham* [63, 12], share the metaphor of viewing a certain kind of distributed state as a "solution" in which many "molecules" float. Concurrent transitions are then viewed as "reactions" that can occur simultaneously in many points of the solution. This metaphor is a suggestive way of describing the case in which the top-level structure of a system's distributed state is a *multiset*. Therefore, both Gamma and the *cham* specify classes of rewrite theories in which the equational axioms $E = ACI$ are the *associativity* and *commutativity* of a multiset union operator $_,_$ having the empty multiset, say λ, as its *identity*.

A Gamma program is essentially a collection of conditional rewrite rules, called *basic reactions*, of the form

$$x_1, \ldots, x_n \longrightarrow A(x_1, \ldots, x_n) \;\; \textit{if} \;\; R(x_1, \ldots, x_n)$$

where the condition R is a boolean expression and A is a multiset expression called the *action*. Typically, concurrent Gamma computations are performed exhaustively until termination is reached.

In the case of the *cham*, there is a common syntax, shared by all chemical abstract machines, that can be expressed as an order-sorted signature Ω. Each *cham* can then be expressed as a rewrite theory $\mathcal{C} = (\Sigma, ACI, L, R)$, with $\Sigma \supseteq \Omega$, together with a partition

$$R = \mathit{Reaction} \uplus \mathit{Heating} \uplus \mathit{Cooling} \uplus \mathit{AirlockAx}.$$

The rules in R are subject to certain syntactic restrictions that guarantee an efficient form of matching modulo ACI. See [115] for some more discussion.

3.6 CCS, LOTOS and the π-Calculus

Kokichi Futatsugi, Timothy Winkler and I [121], and in a different later version Narciso Martí-Oliet and I [105], have shown two different ways in which Milner's CCS can be naturally represented in rewriting logic. One representation essentially treats the transitions as rewrite rules, with some syntactic care to record in the term the actions that have been performed. The other representation considers the operational semantics rules of CCS as the rewrite rules of a rewrite theory and provides a more declarative account. In both of them the representation exactly characterizes the legal CCS computations [105]. Another rewriting specification of CCS in a double category model that is a natural generalization of the 2-category models of rewriting logic has been proposed by Gadducci and Montanari [60] and is discussed in Section 3.13.

LOTOS [72] is a specification language combining the two formalisms of algebraic data types and (an extension of) CCS. It is pointed out in [121] that writing an executable specification of LOTOS in rewriting logic that could be used as a LOTOS interpreter is both very natural and straightforward. In fact, an interpreter of this kind has been written by Futatsugi and his collaborators with very

good results [137]. The point is that the algebraic and process formalisms—whose relationship seems somewhat unclear in their original LOTOS combination—find what might be called their true semantic home in rewriting logic, where the equational part is accounted for by the equational signature and axioms, and the process part is described by rewrite rules over the corresponding expressions. Viry [158] makes essentially the same observation about the naturalness of rewriting logic as a semantic framework for LOTOS, and also points out that the particular syntactic restrictions imposed by LOTOS make the combined execution of LOTOS equations and LOTOS transition rules very easy by rewriting, because they satisfy the *coherence* property defined in [158].

More recently, Viry [159] has given a very natural specification of the π-calculus in rewriting logic. The realization that the operational semantics of the π-calculus can be naturally described using rewrite rules modulo the associativity and commutativity of a multiset union operator goes back to Milner [128]. However, as in the case of rewriting logic specifications of the lambda calculus discussed in Section 3.1, binding operators become an extra feature that should be accounted for. As for the lambda calculus, the answer given by Viry [158] resides in an equational theory of explicit substitution, so that expressions up to alpha-conversion can be regarded as equivalence classes.

3.7 Concurrent Objects, Actors, and OO Databases

In a concurrent object-oriented system the concurrent state, which is usually called a *configuration*, has typically the structure of a *multiset* made up of objects and messages. Therefore, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax as

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

where the multiset union operator `__` is declared to satisfy the structural laws of associativity and commutativity and to have identity `null`. The subsort declaration in the first line above states that objects and messages are singleton multiset configurations, so that more complex configurations are generated out of them by multiset union.

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

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

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

Intuitively, we can think of messages as "traveling" to come into contact with the objects to which they are sent and then causing "communication events" by application of rewrite rules. In rewriting logic, this travelling is accounted for in a very abstract way by the structural laws of associativity, commutativity, and identity. The communication events can be either synchronous or asynchronous. *Asynchronous message passing* communication between objects is typical of *Actor systems* [3, 2]. Generalizing slightly the Actor case, the Simple Maude language [98] adopts the following general form of conditional rules for asynchronous message passing interaction between objects

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

Such rules involve at most one object and one message in their lefthand side, where the notation (M) means that the message M is only an optional part of the lefthand side, that is, that we also allow *autonomous objects* that can act on their own without receiving any messages. Similarly, the notation $(\langle O : F' \mid atts' \rangle)$ means that the object O—in a possibly different state—is only an optional part of the righthand side, i.e., that it can be omitted in some rules so that the object is then deleted. In addition, p new objects may be created, and q new messages may be generated for $p, q \geq 0$.

The above representation of objects is the one adopted in the Maude language [112, 127, 116] and implicitly assumes globally unique names. Other representations are of course possible. For example, in her theory of actors and actor components, Carolyn Talcott adopts instead a more abstract representation of objects as *abstract actors* [153], where objects can be renamed by a form of alpha-conversion to avoid name clashes across components. In two very fine papers [153, 152] she then uses rewriting logic to reason formally about the behavior of actors and of components, and studies their infinite fair computations.

Object interaction need not be asynchronous. It may involve events in which several objects, with or without the prompting of messages, synchronously participate in a local transition. Rewriting logic can easily specify such synchronous interactions between objects as more general rewrite rules of the form

$$
\begin{array}{lll}
(\ddagger) & M_1 \ldots M_n\ \langle O_1 : F_1 \mid atts_1 \rangle \ldots \langle O_m : F_m \mid atts_m \rangle & \\
& \longrightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle & \\
& \quad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle & \\
& \quad M'_1 \ldots M'_q & \\
& \quad \textit{if}\ C &
\end{array}
$$

where the Ms are message expressions, $i_1, \ldots, i_k$ are different numbers among the original $1, \ldots, m$, and C is the rule's condition. As we shall see later, some

particular instances of rules of the form (‡) correspond to the UNITY language, graph rewriting, dataflow, and neural net computations. In cases where an asynchronous implementation is preferable, one can use the method proposed in [98] to transform—under quite general assumptions—synchronous rules of the form (‡) into simpler Actor-like rules of the form (†).

An important problem in concurrent object-oriented programming to which rewriting logic has been successfully applied is the so-called *inheritance anomaly* [110], that is, the serious difficulties often encountered when trying to integrate object-oriented inheritance and concurrency in a programming language. The paper [117] shows how the inheritance anomaly can be resolved by adopting a declarative programming style with rewrite rules. A more recent paper by Lechner, Lengauer, Nickl and Wirsing [93] proposes additional rewriting logic techniques to give a somewhat different solution to this problem.

A number of other authors have developed various object-oriented applications of rewriting logic. For example, Lechner, Lengauer, and Wirsing have carried out an ambitious case study investigating the expressiveness of rewriting logic and Maude for object-oriented specification and have explored refinement concepts in [94]; and Wirsing, Nickl and Lechner [163] have proposed the rewriting logic-based *OOSpectrum* formalism for formal object-oriented specifications. Wirsing and Knapp have studied the important topic of how to pass from more informal specifications expressed in any of the widely accepted object-oriented design notations to formal specifications in rewriting logic [162] (see Section 5.1). From a different, (co-)algebraic, perspective, Reichel has found rewriting logic useful in his final coalgebra semantics for objects [143]. The benefits of rewriting logic for execution of, and formal reasoning about, object-oriented *discrete event simulations* is another application area that has been investigated by Landauer [86].

Yet another area where rewriting logic has proved useful is in the specification and programming of *object-oriented databases*. Meseguer and Qian [124] have shown how the equational approach to object-oriented databases and bulk data types taken by other database researchers can be extended thanks to the use of rewrite rules to deal with the dynamic aspect of database *updates*, so that a formal executable specification of all the aspects of an object-oriented database can be achieved. Denker and Gogolla [44] have used Maude to give semantics to the TROLL *light* object-oriented database specification language; this work has the advantage of providing a formal link between rewriting logic and the algebraic approach to information systems proposed by the IS-CORE Group [146, 77]. More recently, Pita and Martí-Oliet [140] have carried out a thorough case study on the application of Maude to the executable specification of a database model for broadcast telecommunication networks.

3.8 Unity

UNITY [27] is an elegant and important theory of concurrent programming with an associated logic to reason about the behavior of concurrent programs that has been developed by K. Many Chandy and Jayadev Misra. As shown in [115]

the rewriting logic approach to object-oriented systems yields UNITY's model of computation as a special case in a direct way.

The details are given in [115], but the basic idea is straightforward. In essence a UNITY program is a set of multiple assignment statements of the form

$$(\star) \quad x_1, \ldots, x_n := exp_1(x_1, \ldots, x_n), \ldots, exp_n(x_1, \ldots, x_n)$$

where the x_i are declared variables, and the $exp_i(x_1, \ldots, x_n)$ are Σ-terms for Σ a fixed many-sorted signature defined on the types of the declared variables. The intuitive meaning of executing such an assignment is that all the variables x_i are *simultaneously* assigned the values that their corresponding expressions $exp_i(x_1, \ldots, x_n)$ evaluate to.

Such a program exactly corresponds to a rewrite theory specifying the behavior of a system composed of "variable" objects of the form $\langle x : T \mid val : v \rangle$ with T a type, having only one attribute, namely a value v of type T.

Each multiple assignment $(\star)$ yields a corresponding rewrite rule, namely the rule:

$$\langle x_1 : T_1 \mid val : v_1 \rangle \ldots \langle x_n : T_n \mid val : v_n \rangle$$
$$\longrightarrow \langle x_1 : T_1 \mid val : exp_1(\overline{v}) \rangle \ldots \langle x_n : T_n \mid val : exp_n(\overline{v}) \rangle$$

3.9 Graph Rewriting

Different mathematical axiomatizations of graph rewriting have been proposed in the literature. The categorical approach using double or single pushouts has been studied quite extensively [50, 147]. However, for our purposes the most convenient axiomatizations are those in which labeled graphs are axiomatized equationally as an algebraic data type in such a way that graph rewriting becomes rewriting modulo the equations axiomatizing the type. Axiomatizations in this spirit include those of Bauderon and Courcelle [10], Corradini and Montanari [38], and Raoult and Voisin [142].

Taking an object-oriented point of view allows a particularly simple axiomatization of graph rewriting in rewriting logic, similar in some respects—although with some notable differences—to the algebraic axiomatization of Raoult and Voisin [142] where graph rewriting is also understood as multiset rewriting. The basic idea is to consider each node of a labeled graph as an *object* with two attributes, one the data element labeling the node—which can belong to any desired data type of values—and the other an adjacency *list* of node names consisting of the immediate neighbors in the graph, that is, nodes to which the node is directly linked. Grouping them in a class *Node*, they have the form

$$\langle a : Node \mid val : v, links : l \rangle$$

An object with this information is essentially what is called a *hyperedge* in the terminology of graph grammars [39], except that hyperedge labels are defined as *unstructured* atomic elements that cannot be further analyzed, whereas we allow them to be structured data on which a graph rewrite rule can also impose patterns. We therefore treat the commonly occurring case in which all the edges

coming out of a node can be naturally formalized by a single hyperedge; however, our treatment can easily be generalized to deal with several hyperedges with a common source node.

In this object-oriented view, a labeled graph is then understood as a *configuration* of node objects. Of course, as for other object-oriented systems, we require that different node objects should have different names. In addition, for such a configuration to be really a graph, there should be no "dangling pointers," that is, if a node name appears in the list of neighbors of a node, then there must be a node object with that name present in the configuration. Graph rewrite rules are then a special case of synchronous object-oriented rewrite rules (‡) that do not involve any messages and that rewrite configurations that are graphs into other configurations that are also graphs.

We illustrate these ideas with an example borrowed from [98], namely a single graph rewrite rule accomplishing the clustering of a two-dimensional image into its set of connected components. We may assume that the image is represented as a two-dimensional array of points, where each point has a unique identifier different from that of any other point, say a nonzero number, if it is a point in the image; points not in the image have the value 0. Figure 1 shows one such image and its two connected components.

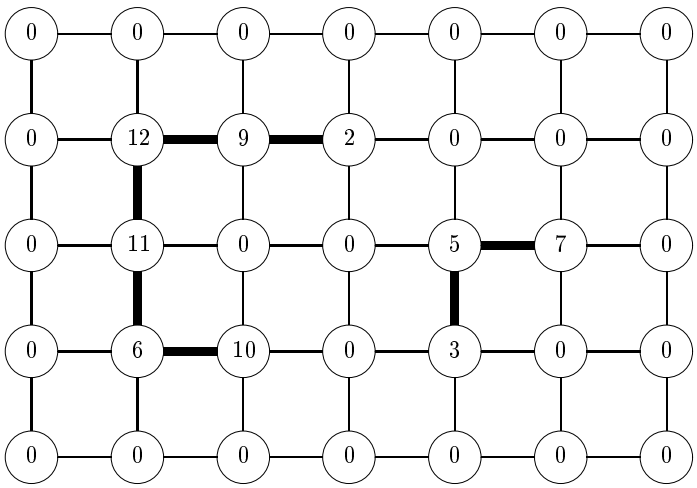


Fig. 1. Image as two-dimensional array.

One way to compute the connected components is to assign to all points in each component the greatest identifier present in the component. In the above example all points in the left component will end up with value 12, and all those

in the right component with value 7. This can be accomplished by repeated application of the single rewrite rule in Figure 2, which can be applied concurrently to the data graph. Note that the rule is conditional on the value N_0 being different from 0. The labels a, b, c, d, e identify the same nodes of the graph before and after the rewrite is performed.

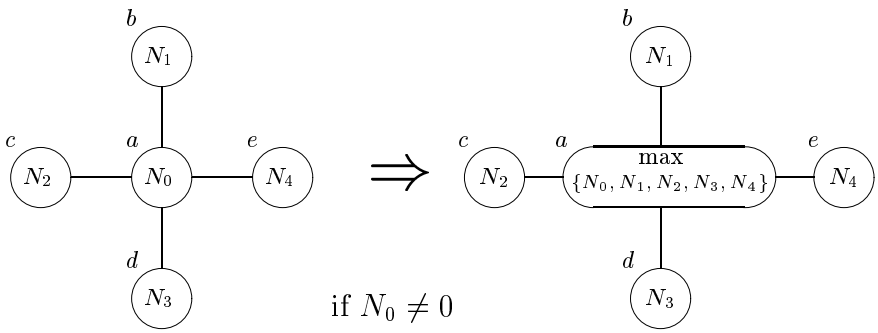


Fig. 2. A graph rewrite rule.

The rewriting logic expression of this rule is straightforward

$$\begin{array}{ll} \langle A : Node \mid val : N_0, links : BCDE\rangle & \rightarrow \langle A : Node \mid val : max\rangle \\ \langle B : Node \mid val : N_1\rangle & \langle B : Node \mid val : N_1\rangle \\ \langle C : Node \mid val : N_2\rangle & \langle C : Node \mid val : N_2\rangle \\ \langle D : Node \mid val : N_3\rangle & \langle D : Node \mid val : N_3\rangle \\ \langle E : Node \mid val : N_4\rangle & \langle E : Node \mid val : N_4\rangle \end{array}$$

$$\begin{array}{c} if\, N_0 \neq 0 \\ where \quad max = max(N_0, N_1, N_2, N_3, N_4) \end{array}$$

See [118] for a more detailed discussion of other topics such as: increasing graph rewriting parallelism through additional equational axioms that allow concurrent rewrites that overlap in "read-only" areas; node creation; and node deletion through garbage collection.

3.10 Dataflow

The dataflow model of computation [75] has been thoroughly investigated both at the level of parallel functional languages, and in terms of parallel architectures directly supporting the dataflow model. The model is very intuitive. One pictures the computation as a graph in which data flows along edges and is computed in nodes labeled by different functions. The actual functions computed by each

node are specified in an associated *data algebra*, which is just an algebraic data type such as the booleans, the integers, or some other many-sorted data type.

The specification of different variants of dataflow in rewriting logic is very direct. The variants in question have to do with the nature of edges. In *piped* models they are FIFO buffers—of arbitrary or of limited capacity—in which data is placed as output of some node and consumed as input to another node. In *tagged* models, edges are instead multisets in which data tagged with a number is placed and consumed. In general, an edge is an object $\langle e : Edge \mid cts : c \rangle$ with just one attribute, namely a list or a multiset data structure.

Nodes are also objects. They contain a list of input edges, one or several output edges, and have a class identifying the functionality of the node. A *state* of the dataflow network is a configuration of edge and node objects. The *firing rules* specifying the concurrent behavior of all networks with the same data algebra exactly correspond to object-oriented rewrite rules in which a node and its input and output edges rewrite together. For example, the firing rule for a functional node computing a function f of n arguments in the piped model is

$$\begin{array}{l} \langle e_1 : Edge \mid cts : x_1 \rangle \ldots \langle e_n : Edge \mid cts : x_n \rangle \\ \langle a : f \mid inputs : e_1 \ldots e_n, output : e \rangle \\ \langle e : Edge \mid cts : l \rangle \\ \quad \longrightarrow \langle e_1 : Edge \mid cts : nil \rangle \ldots \langle e_n : Edge \mid cts : nil \rangle \\ \qquad \langle a : f \mid inputs : e_1 \ldots e_n, output : e \rangle \\ \qquad \langle e : Edge \mid cts : f(x_1 \ldots x_n)\, l \rangle \end{array}$$

where we of course assume that the equations in our rewrite theory axiomatize the data algebra, so as to compute the actual value of $f(x_1 \ldots x_n)$. Variants in which the FIFO buffer is bounded, as well as the tagged case can be handled with similar firing rules. As explained in [118], concurrency can be further increased by allowing simultaneous write and read access to the edges; this can be axiomatized by additional equations.

3.11 Neural Networks

Artificial neural networks [101] are another important model of parallel computation. The basic idea is very simple: they are networks of computing nodes where each node simulates the behavior of a biological neuron. Each connection has a *weight*, and the neuron has a *threshold* θ that determines whether the stimulus is strong enough to cause its firing; both the weights and the threshold can be changed by training. When all the inputs from the connections with other neurons have been received, if their weighted sum exceeds the threshold, they cause the *firing* of the neuron's response, whose actual value is simulated as a particular function of the weighted sum of inputs minus the threshold. The formalization in rewriting logic is very direct. Neurons can be regarded as objects

$$\begin{array}{l} \langle b : Neuron \mid in(a_1) : (w_1, v_1), \ldots, in(a_n) : (w_n, v_n), \\ \qquad\qquad thld : \theta, function : f, out(c_1) : u_1, \ldots, out(c_m) : u_m \rangle \end{array}$$

where the $a_1, \ldots, a_n$ are the neurons providing inputs, the $c_1, \ldots, c_m$ those receiving b's output, f is the name of the function governing the firing, the $w_1, \ldots, w_n$ are the weights of the corresponding input connections, the $v_1, \ldots, v_n$ are either numerical values for the inputs, or the nonnumerical constant mt if a particular input has not been received, and the $u_1, \ldots, u_m$ are either *true* or *false* depending on whether the output has being received or not by the corresponding target neuron. The main rewrite rule is the firing rule for neurons, namely,

$$\begin{aligned}
&\langle b : \mathit{Neuron} \mid \mathit{in}(a_1) : (w_1, x_1), \ldots, \mathit{in}(a_n) : (w_n, x_n), \\
&\qquad\qquad \mathit{thld} : \theta, \mathit{function} : f, \mathit{output} : o, \mathit{out}(c_1) : \mathit{true}, \ldots, \mathit{out}(c_m) : \mathit{true} \rangle \\
&\rightarrow \langle b : \mathit{Neuron} \mid \mathit{in}(a_1) : (w_1, mt), \ldots, \mathit{in}(a_n) : (w_n, mt), \\
&\qquad\qquad \mathit{thld} : \theta, \mathit{function} : f, \mathit{output} : f((\Sigma_i x_i w_i) - \theta), \\
&\qquad\qquad \mathit{out}(c_1) : \mathit{false}, \ldots, \mathit{out}(c_m) : \mathit{false} \rangle
\end{aligned}$$

where the x_i are all variables of numerical type, and where we of course assume that the equational part of the specification fully axiomatizes all the required numerical computations, including the function f. Training of nets can also be easily formalized by similar rewrite rules, but a few more attributes must be added to each neuron for this purpose.

3.12 Real-Time Systems

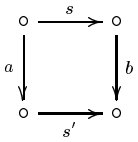
The first important research contribution exploring the application of rewriting logic to real-time specification has been the work of Kosiuczenko and Wirsing on *timed rewriting logic* (TRL) [84], an extension of rewriting logic where the rewrite relation is labeled with time stamps. Axioms in TRL are sequents of the form $t \xrightarrow{r} t'$. Their intuitive meaning is that t evolves to t' in time r. The rules of deduction of standard rewriting logic are extended, and are further restricted, with time requirements, to allow only deductions in which all the different parts of a system evolve in the same amount of time. TRL has been shown well-suited for giving object-oriented specifications of complex hybrid systems such as the steam-boiler [138]. In fact, rewriting logic object-oriented specifications in the Maude language have a natural extension to TRL object-oriented specifications in *Timed Maude* [138].

Although it is in some sense possible to regard rewriting logic as a subcase of TRL in which all rules take zero time, Peter Ölveczky and I have investigated a different alternative, namely, using standard rewriting logic to directly specify real-time systems [139]. The basic idea is to make time explicit, as an additive monoid that acts on the states of the system according to specific rewrite rules. We have shown that a good number of frequently used models of real-time computation—including *timed automata* [6], *timed Petri nets* [156, 129], *timed transition systems* [69], and *hybrid automata* [5, 4]—all have a natural and direct expression in standard rewriting logic. In some models rewrite strategies are not important; in other models it is important to give higher priority to certain rules, to ensure that certain actions will be taken as soon as possible.

3.13 Tile Models

The tile models of Gadducci and Montanari [58, 59] generalize the 2-categorical models of rewriting logic introduced in Section 2.4. We saw that an (unsorted and unconditional) rewrite theory $\mathcal{R}$ gives rise to a 2-category $\mathcal{L}_{\mathcal{R}}$ with objects the natural numbers, horizontal arrows the E-equivalence classes of terms $[t]$, and 2-cells the equivalence classes of proof terms $[\alpha] : [t] \longrightarrow [t']$, with horizontal and vertical 2-cell composition operations. Mathematically, tile models correspond to the generalization from 2-categories to *double categories* [78].

Whereas in an ordinary rewrite theory a rewrite rule $t \longrightarrow t'$ corresponds to a 2-cell from which other more complex 2-cells are generated, a tile rewrite rule has instead the form $s \overset{a}{\underset{b}{\longrightarrow}} s'$. Its computational interpretation is that the configuration s of a system can evolve to the configuration s' producing then an *effect* b, which can be observed by the rest of the system. However, such a step is allowed if and only if the subcomponents of s (which is in general an *open* configuration) evolve to the subcomponents of s', producing the *trigger* a. Therefore, tile rewrite rules add to ordinary rewrite rules a *synchronization condition* expressed by a trigger a and an associated observable effect b. Categorically this corresponds to passing from 2-cells to double cells, and from horizontal and vertical 2-cell composition to horizontal and vertical double cell composition. The name "tile" is justified by the pictorial representation of a tile rewrite rule as a double cell



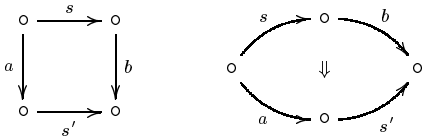
Tile models are very well suited to deal with synchronization aspects of a system, and can be fruitfully applied to a wide variety of synchronous process calculi [59, 22]. Furthermore, they provide a very general categorical framework for structural operational semantics (SOS) definitions of systems, that can be naturally regarded as tile theories satisfying appropriate restrictions that depend on the SOS format adopted [58, 59].

It is intuitively clear that tile rewrite theories generalize standard unconditional rewrite theories. However, two interesting research questions have only recently been answered:

1. Can we map tile logic back into rewriting logic?
2. Model-theoretically, what is the double category analogue of a 2-category with 2-*products* like $\mathcal{L}_{\mathcal{R}}$, or, more generally, what is the right double category analogue of a symmetric monoidal category?

Both questions have been answered in joint work by Bruni, Meseguer, and Montanari, using partial membership algebra [120] techniques. Question 1 has been answered in [123, 22]. Question 2 has been answered in [22], giving rise to two

enriched versions of tile logic, namely *process tile logic*, whose models are symmetric monoidal double categories, and *term tile logic*, whose models are cartesian double categories. In fact, questions 1 and 2 are closely related, because the translation from tile logic into rewriting logic relates also the corresponding double- and 2-category models, bringing them together into a combined structure called a 2-VH category. The key intuition about the translation answering question 1 is that we translate tile rules into ordinary rewrite rules by flattening double cells into 2-cells as follows



An important practical benefit of such a translation is that we can use a rewriting logic language to execute tile logic specifications. Model-theoretically, the restrictions on allowable rewrites imposed by the synchronization conditions expressed by the vertical sides mean that the legal tile logic proofs constitute a *subsort* of all the rewrite proofs. In practice such a subsort can be carved out by using a rewriting *strategy* that selects exactly the tile proofs. Several examples illustrating the use of Maude strategies to implement tile logic specifications are discussed in [22].

4 A Semantic Framework for Programming Languages

On the one hand, rewriting logic can be used as a wide-spectrum language encompassing executable specification and concurrent declarative programming. On the other, many other languages can be semantically defined and executed in rewriting logic. Compilers and partial evaluators can also be so defined and built.

4.1 Rewriting Logic Languages

Several language implementation efforts in Europe, the US, and Japan have adopted rewriting logic as their semantic basis and support either executable rewriting logic specification, or declarative concurrent programming in rewriting logic.

Executable Specification Languages. Rewriting logic is particularly well-suited for the executable specification of systems and languages, including concurrent and distributed ones. As further discussed in Section 6, rewriting logic has also very good properties as a logical framework in which many other formal systems can be naturally represented. Several research groups have developed language tools to support formal reasoning and executable specification in rewriting logic.

The ELAN language is a rewriting logic language developed at INRIA Lorraine by P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau and M. Vittek [79, 160, 20]. It has as modules *computational systems*, consisting of a rewrite theory and a *strategy* to guide the rewriting process [18, 19]. This group and their collaborators have developed a very impressive collection of examples and case studies in areas such as logic programming languages, constraint solving, higher-order substitution, equational theorem-proving and other such computational systems [79, 160, 81, 16, 26, 145, 61, 62]. Besides the ELAN interpreter, there is ongoing work on compilation techniques, including compilation of AC-rewriting [130]. A nice feature of rewriting logic, namely its natural way of dealing with concurrency and interaction, is exploited by Viry to treat input-output for ELAN within the logic itself [159].

The CafeOBJ language implementation, developed at the Japan Advanced Institute of Science and Technology (JAIST) in Kanazawa by K. Futatsugi, R. Diaconescu and T. Sawada [57, 56], which is also based on rewriting logic, contains OBJ as its functional sublanguage, and supports object-oriented specifications. Furthermore, its semantics is multi-logical and includes hidden-sorted versions of equational and rewriting logic [47, 56]. The CafeOBJ language has been the basis of an ambitious research effort—the Cafe Project—involving several research institutions in Japan, Europe and the US, as well as several Japanese industries, to exploit the promising possibilities of rewriting logic for formal methods applications in software engineering [24]. This project has achieved a distributable version of the language and further work on its semantics, a collection of specification libraries and case studies, an environment, and a collection of theorem proving tools supporting different forms of verification. Furthermore, a compiler has been developed in addition to the Cafe interpreter implementation [136, 71].

In our group at SRI, Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln and I have been working on the implementation of an interpreter for Maude [127, 116, 32, 31]. After several experimental versions, a beta release has been finished in March 1998 [31], and a full distribution is planned for the Fall of 1998. The equational logic underlying Maude's rewriting logic is membership equational logic [120, 21]. Maude's *functional modules* are Church-Rosser and terminating equational theories; its *system modules* specify general rewrite theories; and its *object-oriented modules* provide syntactic sugar for object-oriented rewrite theories. These modules can be combined by module composition operations in the OBJ style. Maude's high-performance rewrite engine makes extensive use of advanced semi-compilation techniques and is highly modular and extensible, so that new matching algorithms for rewriting modulo different equational theories can easily be added and can be efficiently combined with those of other theories [51, 31]. In addition, Maude's design and implementation systematically exploit the reflective properties of rewriting logic through its `META-LEVEL` module and the use of internal rewriting strategies. Maude has been used in a wide range of applications that are discussed throughout this paper.

Concurrent Rewriting Logic Languages. Since in general rewriting can take place *modulo* an arbitrary set of structural axioms E, which could be undecidable, some restrictions are necessary in order to use rewriting logic for concurrent programming. In the Maude project we have considered two subsets of rewriting logic. The first subset, in which the structural axioms E have algorithms for finding all the matches of a pattern modulo E, gives rise to the Maude language. A second, smaller subset gives rise to Simple Maude [98], a sublanguage that can be used as a machine-independent parallel programming language. Program transformation techniques can then support passage from general rewrite theories to Maude modules and from them to modules in Simple Maude [98]. Patrick Lincoln, Livio Ricciulli and I have developed parallel compilation techniques and a prototype Simple Maude compiler [99] that generates efficient parallel code for the Rewrite Rule Machine (RRM) [100]. Building upon the reflective and concurrent object-oriented features of Maude and the Simple Maude experience, in the near future we plan to develop Mobile Maude, a rewriting logic language in which both data and code will be able to migrate across the nodes of a network.

At INRIA Lorraine, C. Kirchner and P. Viry [80] have studied parallel implementation techniques for rewriting on loosely coupled parallel machines and have experimented with their techniques through a particular implementation in a transputer-based machine. Their approach addresses the standard term rewriting case, and provides new implementation techniques for this case on multicomputers.

Ciampolini, Lamma, Mello, and Stefanelli at the University of Bologna, have designed a parallel programming language called Distributed Logic Objects (DLO) that corresponds to an adequate subset of object-oriented rewrite theories [28]. They have developed a number of implementation techniques for efficiently executing DLO in multicomputers. In their experience, rewriting logic provides a more attractive approach than stream-based parallel logic programming implementations; they point out that the actor subset of object-oriented rewriting chosen in Simple Maude has also in their experience particularly good features for efficient implementation.

4.2 Executable Programming Language Specification

Rewriting logic is a promising semantic framework for formally specifying programming languages as rewrite theories. Since those specifications can be executed in a rewriting logic language, they in fact become *interpreters* for the languages in question. In addition, such formal specifications allow both formal reasoning and a variety of formal analyses for the languages so specified.

The use of rewrite rules to define the semantics of programming languages is of course not new. In a higher-order version it goes back to the use of semantic equations in denotational semantics; in a first-order version, the power of equational specifications to give semantic definitions of conventional languages has been understood and used for a long time (see for example [66] and [157] for uses of OBJ and of ASF+SDF in equational programming language specification, as

well as the important related work on Action Semantics [131]). However, both the lambda calculus and executable equational specifications implicitly assume that such language definitions can be given in terms of *functions*, and rely on the Church-Rosser property to reach the result of an execution.

For conventional languages, by making the state of the computation explicit, a functional description can always be achieved. The situation becomes quite harder for languages that support highly concurrent and nondeterministic applications, and where the possibly nonterminating *interactions* between processes or components—as opposed to the computation of an output value from given inputs—is the whole point of a program. Such languages and applications do not have a meaningful equational description in terms of functions, but do have a very natural rewriting logic semantics, not only operationally, but also denotationally. In fact, we can take the initial model of the rewrite theory specifying a language as its denotational model. Furthermore, aspects having to do with the particular *strategy* used in a language, that can be essential for its semantics, can be very naturally specified and prototyped by rewrite rules at the metalevel using reflection, as already explained in Section 2.5.

Since structural operational semantics definitions [141] can be used for languages not amenable to a functional description, it is natural to compare them with rewriting logic definitions. Their relationship has been discussed in detail in [105]. In fact, both "big-step" (also called "natural" by Kahn [36]) and "small-step" structural operational semantics definitions can be naturally regarded as special formats of corresponding rewrite theory definitions [105]. Tile models provide yet another systematic way of understanding structural operational semantics definitions as tile rewrite theories [59]. Two advantages of rewriting logic specifications are their systematic support for structural axioms—yielding more abstract and compact definitions—and the existence of a model-theoretic semantics, that in fact provides a "true concurrency" model for the computations of a language so specified.

Although the potential of rewriting logic for programming language definitions has yet to be fully exploited in practice, a number of encouraging case studies have already been carried out by different authors. Firstly, some of the models of computation discussed in Section 3 are so closely connected with languages that their rewriting logic specifications are also language specifications. Good examples are rewriting logic definitions of the lambda calculus and (mini-) ML [88, 105], CCS (see [121, 105], the work of Bruni and Clavel [29, 22], and the tile rewrite rules in [59]) the π-calculus [159], and sketches of UNITY [115] and Gamma [118]. Secondly, the usefulness of rewriting strategies to specify program evaluations has been clearly demonstrated in ELAN specifications for Prolog and for the functional-logical language Babel [160], and also in the Bruni-Clavel Maude specification for CCS [29, 22]. Thirdly, the fact that rewriting logic naturally supports concurrent objects has proved very useful in formally specifying a number of novel concurrent languages. For example, Ishikawa et al. [70] have given a Maude specification of a representative subset of the GAEA language [135], a reflective concurrent logic programming language for AI applica-

tions based on a new paradigm called organic programming. Another important example studied by Mason and Talcott [109] building upon Talcott's work on actor theories [152], is the use of rewriting logic to give semantic definitions of actor languages, and to "compile away" certain language features by defining semantics-preserving translations between actor languages that are formalized as translations between their corresponding rewrite theories [109].

A closely related area that seems ripe for the application of rewriting logic techniques and tools is compilation, supercompilation, and partial evaluation. Compilation involves the application of a series of transformation rules to an adequate representation of a program to obtain its compiled form. Such transformation rules can be naturally understood as rewrite rules. However, the rules need not be Church-Rosser; when they should or should not be applied is a delicate matter affecting the result of the compilation process. This means that in general they are not equations, but rules in a rewrite theory; and that they must be applied with a specific *strategy*. Therefore, rewriting logic can be used to formally specify and prototype compilers, and in fact to build them. The process of *driving* at the heart of supercompilation [154] and of partial evaluation [76] is a different name for *narrowing* [54], and is therefore intimately connected with rewriting; furthermore, the reflective aspects involved in supercompilation and partial evaluation are intimately connected with rewriting logic reflection. An interesting first experiment has been the Maude executable specification of the Refal supercompiler [155, 154] by Clavel and Turchin; many more applications lie ahead.

5 A Semantic Framework for Software Architectures

The design, development, and maintenance of a system can be greatly helped by good documentation of its *architecture*, that is, of how it is structured into meaningful subsystems or *components*, and how those subsystems are glued together to form the overall system. In fact, it is very useful to document a system's architecture as early as possible during its design, and to subject such architectural designs to different kinds of analysis and validation techniques to identify errors and inconsistencies as early as possible, before substantial implementation efforts are wasted on realizing the wrong design.

It is therefore very useful to capture a system's architecture in some sufficiently precise notation or language that can then be used as a means of communication between software engineers and as a machine-tractable basis for various forms of design analysis and validation. Languages of this kind are called *architecture description languages* (ADLs) (see [149] for a good introduction to software architectures and ADLs). Some such languages are graphical notations that describe the components of a system and their interconnections with boxes and arrows, but of course many different notations, or, as they are called, many different *architectural styles*, graphical or not, are possible. One particular subfamily is that of object-oriented design notations, an area where there has been a recent effort to unify such notations in the Unified Modeling Language (UML)

of Booch, Jacobson and Rumbaugh [55].

All of this is very useful, but in the present state of the software architecture field several important questions have not yet been answered. One important question is the formal *semantics* underlying a particular ADL. Although some ADLs have been defined with a serious concern for making this very precise, others remain informal notations. The importance of having a formal semantics is not, as it might appear, an academic exercise; it is eminently practical. Without such a semantics an architectural notation becomes badly crippled, because it becomes impossible to subject the designs it expresses to any meaningful *formal analysis*; that is, any application of formal methods is out of the question. One way of understanding why this is bad in practice is to realize that *if* one had a formal semantics for a design in the form of an *executable specification* one could in fact execute such a design, finding many bugs and inconsistencies early in the design process.

Another important question is how to *interoperate* different ADLs. The need for interoperation is related to the fact that different subsystems may have widely different architectural structures, and therefore they may be best expressed in different architectural styles and languages. Of course, the need for a formal semantics reappears in an even more intense form, because now it is not even enough to have a semantics for each of the ADLs. It becomes imperative to systematically relate those semantic accounts in order to achieve a semantic form of interoperation supporting, for example, the execution of the whole design by a combined execution of executable specifications for each of the subsystems.

Carolyn Talcott and I have proposed rewriting logic as a promising semantic framework for ADLs and their interoperation [126] that provides an answer to the above two questions about how to give ADLs a formal semantics and how to interoperate them. We have illustrated how rewriting logic can be used in this way by means of a nontrivial case study involving several architectural styles such as: pipes-and-filters, dynamic architectures, implicit invocation, and message passing, and we have shown how the corresponding rewriting logic specifications can be executed and interoperated in Maude. The essential reason why rewriting logic offers a simple solution to these problems is—as already discussed in Section 3—its flexibility and generality to express widely different models of computation. Typically, each architectural style has a different underlying model of computation as its semantic basis: for example, a pipes-and-filters style has a dataflow model, a message passing model has an asynchronous object-oriented model, and so on. Designs in each of these styles and models then define rewrite theories that can be easily combined and executed in a resulting overall rewrite theory.

5.1 From Object-Oriented Design to Rewriting Logic

M. Wirsing and A. Knapp [162] have argued lucidly about the practical importance of combining object-oriented design notation with formal executable specifications. Their proposal is to annotate diagrams in an existing and widely-used object-oriented notation and then derive formal executable rewriting logic

specifications written in an extension of Maude. The specifications can then be used to execute and formally analyze the designs, and even to generate code in a conventional language such as Java. The importance of combining object-oriented design notations with formal specifications has also been acknowledged in the most mainstream modeling language, namely UML, that has an explicit place holder for such specifications as annotations.

To make their proposal very concrete and illustrate it in practice with examples, Wirsing and Knapp have adopted in [162, 83] Jacobson's OOSE method and notation [74], and have shown in detail how formally annotated OOSE diagrams can be semi-automatically transformed into Maude or CafeOBJ specifications. Furthermore, they have illustrated how this works in a concrete case study. The particular choice of OOSE is not essential to their main ideas and approach; it seems clear that UML—not yet available when their paper was published—could have been used in an entirely similar way.

A related proposal is the GILO-2 scenario-based object-oriented design notation proposed by S. Nakajima and K. Futatsugi [134]. In GILO-2, scenario-based object-oriented diagrams are systematically transformed into rewriting logic formal specifications in CafeOBJ. In fact, rewriting logic provides a formal operational semantics for GILO-2 designs, and allows their execution and formal analysis.

A promising research direction is using reflection to reify the passage from object-oriented diagrams to rewriting logic specifications within rewriting logic itself. This could be accomplished using the general method sketched in Section 2.5; that is, object-oriented diagrams, perhaps with some annotations, can be viewed as a language $\mathcal{O}$, and the passage to their corresponding rewriting logic specifications can be understood as a mapping

$$\Phi : \mathcal{O} \longrightarrow \mathit{RWLogic}.$$

Using reflection we could then reify such object-oriented diagrams as an algebraic data type $\texttt{Module}_{\mathcal{O}}$, and the mapping Φ as an equationally-defined function

$$\overline{\Phi} : \texttt{Module}_{\mathcal{O}} \longrightarrow \texttt{Module}.$$

All this could of course be defined and executed in a reflective rewriting logic language such as Maude, and could be the core of a tool supporting the combination of object-oriented designs and rewriting logic specifications.

5.2 Rewriting Semantics of Software Components

One of the most important recent developments in software technology is the emergence of standards and languages for the distributed interoperation of software *components*. At stake is the passage from mostly monolithic and hard-to-evolve software systems to much more flexible and cost-effective open distributed systems built by gluing together well-documented components provided by an emergent software component industry. Several standards and products—including OMG's CORBA, Microsoft's DCOM, and Sun's Java Beans—are currently competing to become the prevailing standard. This is an important area

in need of semantic foundations that identify the essential aspects of open distributed computing in a language-independent way. Because of its good support for objects, distribution, and reflection, rewriting logic seems a promising formalism for this task.

Three recent papers by Talcott [152], Najm and Stefani [132], and Nakajima [133] have proposed using rewriting logic as a semantic foundation for components and their open distributed computation. Talcott's paper [152] gives a rewriting logic semantics for actor systems that are axiomatized by actor theories. She then extends actor systems by an algebra of *components*, where a component is formalized as a distributed subsystem that can include actors, messages, and other (sub-)components and that is encapsulated by a precisely defined interface that mediates its interactions with an open environment. Najm and Stefani [132] have used rewriting logic to give a formal semantics to the operational subset of the Reference Model for Open Distributed Processing (RM-ODP) [73]. RM-ODP specifies a language-independent computational model underlying many of the distributed computational models that have been recently proposed, including those adopted for CORBA, the Network Object system, and the Java RMI system. Nakajima [133] shows how the calculus of mobile ambients of Cardelli and Gordon [25] can be naturally expressed in rewriting logic, and then uses rewriting logic specifications in CafeOBJ to formally specify the architecture of WEB-NMS, a Java/ORB implementation of a network management system with four architectural components.

6 Logical Framework Applications

Rewriting logic is like a coin with two inseparable sides: one computational and another logical. A proof term is a concurrent computation and viceversa. The generality and expressiveness of rewriting logic as a semantic framework for concurrent computation has also a logical counterpart. Indeed, rewriting logic is also a promising *logical framework* or *universal logic* in which many different logics and formal systems can be naturally represented and interrelated.

6.1 Representing Logics

The basic idea is that we can represent a logic $\mathcal{L}$ with a finitary syntax and inference system within rewriting logic by means of a representation map

$$\Phi : \mathcal{L} \longrightarrow \mathit{RWLogic}.$$

The map Φ should preserve and reflect theoremhood, that is, it should be a *conservative* map of entailment systems in the sense of the theory of general logics [111]. The reason why rewriting logic is a good framework—that is, why it is easy to define maps Φ of this kind for many logics—is that the formulas of a logic $\mathcal{L}$ can typically be equationally axiomatized by an equational theory, and the rules of inference can be typically understood as rewrite rules, that may be conditional if the inference rules have "side conditions." Therefore, the mappings

Φ are usually very simple and direct. Furthermore, as already mentioned in Section 2.5, using reflection we can define and execute maps Φ of this kind inside rewriting logic itself getting an equationally defined map

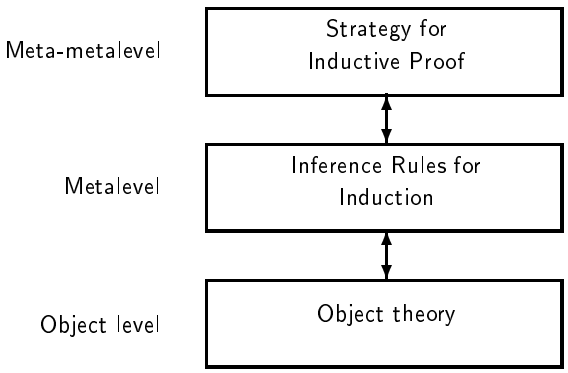
$$\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \longrightarrow \texttt{Module}.$$

In a reflective language like Maude—where a term in `Module` can be entered as a module into Maude's rewrite engine and can then be used to rewrite expressions—this automatically gives us an implementation of the logic $\mathcal{L}$ in rewriting logic. A good number of examples of logic representations can be found in two joint papers with Martí-Oliet [105, 106]. In particular, the map $\overline{\Phi}$ internalizing a representation map Φ from linear logic to rewriting logic was sketched in [105] and has been implemented in Maude by Manuel Clavel in [29].

6.2 Specifying and Building Formal Methods Tools

Theorem provers and other formal methods tools have underlying *inference systems* that can be naturally specified and prototyped in rewriting logic. Furthermore, the strategy aspects of such tools and inference systems can then be specified by rewriting strategies. The researchers in the ELAN group have developed an impressive collection of rewriting logic specifications for different automated deduction inference systems, including logical languages, unification and narrowing [79, 160], Knuth-Bendix completion with constraints [81], higher-order unification [16], combination of unification algorithms [145], constraint solving [26], and termination and tree-automata techniques [61, 62]. In a somewhat similar vein, the work of Levy and Agustí [96, 95, 97] and of Schorlemmer [148] explores the relationships between rewriting logic and their general bi-rewriting approach to automated deduction.

Using a high-performance reflective rewriting logic language like Maude it is possible to turn a theorem prover prototype into an implementation with competitive performance. This has been demonstrated for two theorem proving tools that we have developed within the Cafe Project, namely an inductive theorem prover for membership equational logic, and a Church-Rosser checker for order-sorted equational specifications [30]. Furthermore, reflection allows a very simple and modular design for such tools. Consider, for example, the design of the inductive theorem prover. Its purpose is to prove inductive properties of a Cafe or Maude functional module T, which has an initial algebra semantics. The theory T about which we want to prove inductive properties is at the object level. The rules of inference for induction can be naturally expressed as a rewrite theory $\mathcal{I}$. But since this rewrite theory uses T as a data structure—that is, it actually uses its representation $\overline{T}$—the theory $\mathcal{I}$ should be defined at the metalevel. Proving an inductive theorem for T corresponds to applying the rules in $\mathcal{I}$ with some *strategy*. But since the strategies for any rewrite theory belong to the metalevel of such a theory, and $\mathcal{I}$ is already at the metalevel, we need *three levels* to clearly distinguish levels and make our design entirely *modular*, so that, for example, we can change the strategy without any change whatsoever to the inference rules in $\mathcal{I}$. This is illustrated by the following picture



The inductive Church-Rosser checker tool has a similar reflective design. Again, the module T, that we want to check is Church-Rosser, is at the object level. An inference system $\mathcal{C}$ for checking the Church-Rosser property uses $\overline{T}$ as a data structure and therefore is a rewrite theory at the metalevel. Since the checking process can be described in a purely functional way, there is no need in this case for an additional strategy layer at the meta-metalevel: two levels suffice. Of course, this method of designing and building theorem proving tools is not restricted to equational reasoning tools but is instead entirely general.

6.3 Metaprogramming

"Metaprogramming" is the capacity of defining programs that operate on other programs as their data—in our case, equational and rewrite theories that operate on other such theories as their data. By observing that, using rewriting logic reflection, we can reify not only theories, but also views—that is, theory interpretations—between them, this includes the more traditional "parameterized programming" capabilities in the Clear-OBJ tradition [23, 67] as a particular instance. The difference is that in that tradition theories are metalevel entities not accessible at the object level of the logic, since this is only possible in an explicitly reflective logical context.

What reflection accomplishes is to *open up to the user* the metalevel of the language, so that instead of having a fixed repertoire of parameterized programming operations we can now define a much wider range of theory-transforming and theory-combining operations that could not be defined using more traditional means. We can think of a parameterized programming operation as a function[7] F that, given a module M, returns a module $F(M)$. This function can then be reified as as user-defined function

$$\overline{F} : \texttt{Module} \longrightarrow \texttt{Module}$$

[7] Of course, such a function could take not one, but several modules as well as additional parameters, that could be subject to additional constraints, but the simpler case described here suffices for our purposes.

inside rewriting logic. For example, an interesting collection of user-definable parameterized constructions of this kind has been used in the design of the Cafe theorem proving tools to perform a variety of useful module transformations [30]. Of course, we can be more general and not restrict the function F to have rewriting logic modules as arguments. For example, the reification of the logic map

$$\Psi : \mathit{LLogic} \longrightarrow \mathit{RWLogic}$$

representing linear logic into rewriting logic gives us an equationally defined function [105, 29]

$$\overline{\Psi} : \mathtt{Module}_{\mathit{LLogic}} \longrightarrow \mathtt{Module}$$

that is a metaprogram in this more general sense.

6.4 Avoiding the Frame Problem

Since rewriting logic is a logic of change whose subject matter is precisely the dynamic changes in context within a system, all the insoluble problems and absurdities that one runs into when trying to formalize change with essentially static logics—the so-called *frame problem*—do not cause any trouble for rewriting logic. Martí-Oliet and I have explained the advantages of rewriting logic for formally representing change, and have illustrated those advantages with many examples. Furthermore, using mappings that represent the formalisms in question within the rewriting logic framework, we have shown how several other logical approaches to dynamic change can be naturally subsumed [108].

6.5 Towards Formal Interoperability

The fact that rewriting logic is a reflective logical framework that can be mechanically supported by language implementations opens up promising possibilities to make solid progress towards the goal of achieving *formal interoperability* [119]; that is, of developing new methods and meta-tools to move in a mathematically precise way across different formalizations, and to use in a rigorously integrated way the different tools supporting such formalizations. In particular, the capacity of defining and executing conservative *mappings* between different logics within the framework of rewriting logic can be of great practical use to rigorously interoperate and combine different formal tools supporting different logics. In fact, the reification technique explained in Section 2.5 can be applied not only to representation maps into rewriting logic, but also to general maps

$$\Phi : \mathcal{L} \longrightarrow \mathcal{Q}$$

between two logics $\mathcal{L}$ and $\mathcal{Q}$, that can then be reified as equationally-defined functions

$$\overline{\Phi} : \mathtt{Module}_{\mathcal{L}} \longrightarrow \mathtt{Module}_{\mathcal{Q}}$$

within rewriting logic. That is, rewriting logic provides an executable framework to implement key concepts in the theory of general logics [111].

There is, in addition, a very fruitful relationship between rewriting logic and the theory of *reasoning theories* proposed by Giunchiglia, Pecchiari and Talcott [64]. Reasoning theories provide a logic-independent architecture for combining and interoperating different mechanized formal systems. They are closely related to rewrite theories and there are fruitful synergies between both concepts that Carolyn Talcott and I are currently investigating [125]. Much work remains ahead, but the prospects of developing useful methods for formal interoperability are encouraging.

7 Formal Reasoning, Refinement, and Transformation

A rewriting logic specification $\mathcal{R}$ provides a mathematical model of the system it specifies, namely the initial model $\mathcal{T}_{\mathcal{R}}$. One can then reason mathematically about such a model to prove important system properties. A number of studies have shown how this can be done. For example, Talcott has used rewriting logic specifications to prove properties of actor systems [153] and, in joint work with Mason, of semantics-preserving translations between actor languages [109]. Similarly, Krogdahl and Lysne [85] have specified and proved a key part of a cache coherence protocol by specifying it in rewriting logic, and have carried out a significant portion of their proof mechanically by rewriting. Communication protocols, including secure ones, are another promising area for rewriting logic specification and formal analysis. They can be very naturally specified and executed as concurrent object-oriented systems that communicate through message passing. Furthermore, the corresponding executable specifications can be formally analyzed by a form of model-checking that explores all the possible behaviors beginning with a given initial state. This can be accomplished by a rewriting strategy that considers all the possible rewritings from that state in all possible orders. Using these techniques, Denker, Meseguer and Talcott [45] have specified the Needham-Schroeder cryptographic protocol plus an attacker object, and have put in evidence a well-known attack. Similarly, Ölveczky, Denker, García-Luna, Meseguer, Smith and Talcott have found a number of flaws and deadlocks in the informal specification of a reliable broadcast protocol (see Appendix B in [31]).

In the process of proving properties about a specification one often wants to begin with a more abstract version and then refine it or transform it; this is discussed in Section 7.1. Furthermore, as discussed in Section 7.2, to prove more global system properties one may find it useful to introduce a property-oriented logical language such as a modal or temporal logic.

7.1 Refinement and Transformation

The refinement of rewriting logic specifications supports program transformations, implementations of more abstract levels by more concrete ones, and important theory composition operations such as the instantiation of parameterized modules. An approach to refinement of rewrite theories by means of maps

$\mathcal{R} \longrightarrow \mathcal{Q}$ that can be best understood as 2-functors $\mathcal{L}_{\mathcal{R}} \longrightarrow \mathcal{L}_{\mathcal{Q}}$ between the corresponding Lawvere 2-theories was proposed in [114]. A similar notion has also been proposed and used by Lechner, Lengauer and Wirsing in [94], and has been further extended by Lechner to deal also with refinement of property-oriented specifications in her modal μ-calculus [91]. Yet another recent development that seems promising for reasoning about behavioral satisfaction and that may provide more flexible ways of refining rewriting logic specifications is Diaconescu's notion of *hidden sorted* rewriting logic [47].

Using completion methods, P. Viry has developed a very useful program transformation technique for rewrite theories [158]. His key notion is that of *coherence* between the equational part E and the rules R of a rewrite theory. This property makes very easy the implementation of such a theory by rewriting techniques without having to have an E-matching algorithm. Lincoln, Martí-Oliet and I have studied several program transformation techniques, including coherence completion, to pass from rewrite theories to theories implementable in Maude, and from Maude specifications to efficient parallel programs in Simple Maude [98].

7.2 Modal and Temporal Logics

Rewriting logic is primarily a logic *of* change—in which the deduction directly corresponds to the change—as opposed to a logic to talk *about* change in a more indirect and global manner such as the different variants of modal and temporal logic. Such logics regard a system as a mathematical model—typically some kind of Kripke structure—about which they then make assertions about its global properties such as safety or liveness.

Both levels of description and analysis are useful in their own right; in fact, they complement each other: one can use both logics in combination to prove system properties. Such a combined approach could be quite powerful, because we can have the most adequate logic to reason about each level, and their combination to support auxiliary reasoning at another level. In fact, under closer analysis, rewriting logic itself is a two-level logic, combining an abstract data type specification level for *static* system aspects—namely, its underlying equational logic—with a logic of change for *dynamic* system aspects—namely rewriting logic. A modal or temporal logic then provides a logic to reason about more global system *properties*.

The first issue that must be addressed is the semantic integration between rewriting logic and the modal or temporal logic of choice. Essentially, such an integration is straightforward, because both logics are talking about the same mathematical model. As explained in Section 2.3, the initial model $\mathcal{T}_{\mathcal{R}}$ of a rewrite theory $\mathcal{R}$ is a category with algebraic structure, where the objects correspond to system states, and the arrows correspond to concurrent system transitions. Therefore, $\mathcal{T}_{\mathcal{R}}$ can be regarded as a Kripke structure whose transitions are labeled by the arrows of the category. A variety of different modal or temporal logics can then be chosen to make assertions about such a Kripke structure.

Since object-oriented systems constitute a particularly wide and important application area, modal or temporal logics that provide explicit support for object systems and can reason about their rewriting logic specifications are of great interest. Two candidate formalisms of this kind have recently been proposed. One is a version of the modal μ-calculus proposed by Ulrike Lechner [92, 90, 91] for reasoning about object-oriented Maude specifications. Another is Denker's object-oriented distributed temporal logic DTL$^+$ [45, 42], that extends the DTL and D_1 distributed object temporal logics of Ehrich and Denker [49, 43, 48].

Lechner [90, 91] uses her version of the modal μ-calculus to identify a number of safety properties of Maude object-oriented specifications that can be characterized as special classes of formulas. She has also done a fine analysis of how certain reuse properties such as inheritance, subconfiguration, and message combination can be reflected at the modal μ-calculus level by the preservation of certain classes of formulas. Furthermore, she has verified properties of useful examples using her logic, and has shown how modal μ-calculus specifications can be *refined* into more specific ones until reaching a rewriting logic specification.

Denker's DTL$^+$ [45, 42] builds upon her previous work on temporal logic specification of object-oriented distributed information systems and has the following aims: (1) DTL$^+$ offers several communication concepts, including synchronous and asynchronous communication; and (2) DTL$^+$ is designed in such a way that the principles of object-oriented specifications based on rewriting logic are captured. In particular, DTL$^+$ can be used to reason about properties of object-oriented Maude specifications. In DTL$^+$ the underlying rewriting logic is extended by temporal operators.

One attractive prospect is the use of Maude's reflective features to build a tool mechanizing the modal or temporal logic deduction and to integrate it with the underlying rewriting logic and equational logic deductions and their supporting formal tools. This would be a very practical application of rewriting logic as a logical framework. At the equational logic level this idea has already been successfully exploited to build an inductive theorem prover and a Church-Rosser checker written in Maude [30]; we now need to develop similar tools for the rewriting logic and temporal logic levels.

8 Concluding Remarks

Thanks to the important contributions of the researchers mentioned in this survey, the rewriting logic research program has advanced to a stage in which more ambitious future tasks can be contemplated. Rewriting logic languages have now quite mature interpreter implementations and promising compiler tools; these will play a key role in future developments. We can expect to see important new advances in moving from interpreter implementations to compilers and to distributed implementations.

The semantic framework applications to specify models of concurrency are quite well developed; but specification, symbolic simulation, and verification of hardware systems—especially dealing with their concurrency aspects—is an area

little explored so far, and wide open for future research. Semantic definition of programming languages and compilers, and using the methods of rewriting logic to support other language definition methodologies should also be a very active area, and should be supported by adequate meta-tools. Applications to software architecture, object-oriented design, and software components have recently started; they all look very promising.

Similarly, even though the meta-theoretic foundations of the logical framework applications are well developed, we may expect many more practical applications in this area. The issue of formal reasoning tools supporting work in all these areas is also of paramount importance. In comparison to the language implementation work, on which it relies, we are clearly less advanced and much more work needs to be done. But the encouraging experience already obtained suggests that building formal tools for rewriting logic, for modal and temporal logics, and for many other logics and inference systems—including symbolic algebra systems—is going to be much easier using rewrite rules and reflective techniques than it would be using conventional means.

With languages, tools and methods sufficiently developed, it will become possible to bring these ideas into contact with industrial practice, both for formal methods applications, and for the development of entire software solutions and tools. Areas such as parallel and distributed programming, mobile computing, communication protocols, active networks, component software, hardware verification, parallel symbolic simulation, software design, compilers, program analysis tools, and programming environments should provide very good application opportunities.

Acknowledgments

I very much wish to thank all the researchers who have contributed through their work to the advancement of these ideas. With many of them I have exchanged ideas in person that have enriched and influenced my views, and from all of them I have learned much. I owe a special debt of gratitude to those researchers with whom I am working or have worked most closely on these topics. They include Adel Bouhoula, Roberto Bruni, Manuel Clavel, Grit Denker, Francisco Durán, Steven Eker, Kokichi Futatsugi, Hiroshi Ishikawa, Jean-Pierre Jouannaud, Patrick Lincoln, Narciso Martí-Oliet, Ugo Montanari, Peter Ölveczky, Xiaolei Qian, Livio Ricciulli, Carolyn Talcott, and Timothy Winkler. I have also benefited much from conversations with Joseph Goguen, Claude and Hélène Kirchner, and Martin Wirsing. Grit Denker and Narciso Martí-Oliet deserve special thanks for their careful reading of earlier drafts of this paper. I finally wish to thank the organizers of the 1997 Marktoberdorf Summer School on Computational Logic for giving me the opportunity of lecturing there on these topics, and for the most enjoyable scientific and human experience that this opportunity afforded.

References

1. P. Aczel. A general Church-Rosser theorem. Manuscript, University of Manchester, 1978.
2. G. Agha. *Actors*. MIT Press, 1986.
3. G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, 1988.
4. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
5. R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R. Grossman, A. Nerode, A. Ravn, and H. Rischel, editors, *Workshop on Theory of Hybrid Systems*, pages 209–229. Springer LNCS 739, 1993.
6. R. Alur and D. Dill. The theory of timed automata. In J. de Bakker, G. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, 1991.
7. A. Asperti. A logic for concurrency. Unpublished manuscript, November 1987.
8. J.-P. Banâtre and D. L. Mètayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.
9. E. Battiston, V. Crespi, F. D. Cindio, and G. Mauri. Semantic frameworks for a class of modular algebraic nets. In M. Nivat, C. Rattray, T. Russ, and G. Scollo, editors, *Proc. of the 3rd International AMAST Conference*, Workshops in Computing. Springer-Verlag, 1994.
10. M. Bauderon and B. Courcelle. Graph expressions and graph rewriting. *Math. Systems Theory*, 20:83–127, 1987.
11. J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer-Verlag, 1980. LNCS, Volume 81.
12. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
13. E. Best and R. Devillers. Sequential and concurrent behavior in Petri net theory. *Theoretical Computer Science*, 55:87–136, 1989.
14. M. Bettaz and M. Maouche. How to specify nondeterminism and true concurrency with algebraic term nets. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, pages 164–180. Springer LNCS 655, 1993.
15. M. Bettaz and M. Maouche. Modeling of object based systems with hidden sorted ECATNets. In *Proc. of MASCOTS'95, Durham, North Carolina*, pages 307–311. IEEE, 1995.
16. P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartosek, J. Staudek, and J. Wiedermann, editors, *Proc. SOFTSEM'95*, pages 363–368. Springer LNCS 1012, 1995.
17. P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.

18. P. Borovansky, C. Kirchner, and H. Kirchner. Strategies and rewriting in ELAN. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, 1997.
19. P. Borovansky, C. Kirchner, and H. Kirchner. Strategies of ELAN: meta-interpretation and partial evaluation. In *Proceedings of the International Workshop on Theory and Practice of Algebraic Specifications (Amsterdam, Holland)*, 1997.
20. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
21. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer-Verlag, 1997.
22. R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Manuscript, March 1998, SRI International and C.S. Dept., Univ. of Pisa, 1998.
23. R. Burstall and J. Goguen. The semantics of Clear, a specification language. In D. Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer LNCS 86, 1980.
24. CafeOBJ-Project. *Proceedings of the CafeOBJ Symposium'98, Numazu, Japan.* April 1998.
25. L. Cardelli and A. Gordon. Mobile ambients. Technical report, Digital Equipment Corporation, Systems Research Center, 1997.
26. C. Castro. An approach to solving binary CSP using computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
27. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
28. A. Ciampolini, E. Lamma, P. Mello, and C. Stefanelli. Distributed logic objects: a fragment of rewriting logic and its implementation. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
29. M. Clavel. Reflection in general logics, rewriting logic, and Maude. Ph.D. Thesis, University of Navarre, 1998.
30. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational logic tools by reflection in rewriting logic. In *in Proc. of the CafeOBJ Symposium '98, Numazu, Japan.* CafeOBJ Project, April 1998.
31. M. Clavel, F. Durán, S. Eker, J. Meseguer, and P. Lincoln. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.
32. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
33. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

34. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
35. M. Clavel and J. Meseguer. Internal strategies in a reflective logic. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, pages 1–12, 1997.
36. D. Clément, J. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. In K. Fuchi and M. Nivat, editors, *Proceedings, France-Japan AI and CS Symposium*, pages 49–89. ICOT, 1986. Also, Information Processing Society of Japan, Technical Memorandum PL-86-6.
37. A. Corradini, F. Gadducci, and U. Montanari. Relating two categorical models of term rewriting. In J. Hsiang, editor, *Proc. Rewriting Techniques and Applications, Kaiserslautern*, pages 225–240, 1995.
38. A. Corradini and U. Montanari. An algebra of graphs and graph rewriting. In D. P. et al., editor, *Category Theory and Computer Science*, pages 236–260. Springer LNCS 530, 1991.
39. B. Courcelle. Graph rewriting: an algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 193–242. North-Holland, 1990.
40. P. Degano, J. Meseguer, and U. Montanari. Axiomatizing net computations and processes. In *Proc. LICS'89*, pages 175–185. IEEE, 1989.
41. P. Degano, J. Meseguer, and U. Montanari. Axiomatizing the algebra of net computations and processes. *Acta Informatica*, 33:641–667, 1996.
42. G. Denker. DTL$^+$: A Distributed Temporal Logic Supporting Several Communication Principles. Technical Report , SRI International, Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA 94025, 1998. *To appear.*
43. G. Denker and H.-D. Ehrich. Specifying Distributed Information Systems: Fundamentals of an Object-Oriented Approach Using Distributed Temporal Logic. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97), Volume 2, IFIP TC6 WG6.1 Intern. Workshop, 21-23 July, Canterbury, Kent, UK*, pages 89–104. Chapman & Hall, 1997.
44. G. Denker and M. Gogolla. Translating TROLL *light* concepts to Maude. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *LNCS*, pages 173–187. Springer-Verlag, 1994.
45. G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998.
46. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. North-Holland, 1990.
47. R. Diaconescu. Hidden sorted rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
48. H.-D. Ehrich. Object Specification. Informatik-Bericht 96–07, TU Braunschweig, 1996.
49. H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for*

Databases and Information Systems, pages 167–198. Kluwer Academic Publishers, 1998.
50. H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Graph Grammars and their Application to Computer Science*. Springer LNCS 532, 1991.
51. S. Eker. Fast matching in combination of regular equational theories. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
52. J. Engelfriet, G. Leih, and G. Rozenberg. Parallel object-based systems and Petri nets, I and II. Technical Report 90-04,90-05, Dept. of Computer Science, University of Leiden, February 1990.
53. J. Engelfriet, G. Leih, and G. Rozenberg. Net-based description of parallel object-based systems, or POTs and POPs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, Noordwijkerhout, The Netherlands, May/June 1990*, pages 229–273. Springer LNCS 489, 1991.
54. M. Fay. First-order unification in an equational theory. In *Proceedings of the 4th Workshop on Automated Deduction*, pages 161–167, 1979.
55. M. Fowler and K. Scott. *UML Distilled*. Addison-Wesley, 1997.
56. K. Futatsugi and R. Diaconescu. CafeOBJ report. To appear in the AMAST Series, World Scientific, 1998.
57. K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proc. of the Kunming International CASE Symposium, Kunming, China, November*, 1994.
58. F. Gadducci. *On the algebraic approach to concurrent term rewriting*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1996.
59. F. Gadducci and U. Montanari. The tile model. To appear in G. Plotkin, C. Stirling and M. Tofte, eds., *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press. Also, TR-96-27, C.S. Dept., Univ. of Pisa, 1996.
60. F. Gadducci and U. Montanari. Enriched categories as models of computation. In *Proc.* 5^{th} *Italian Conference on Theoretical Computer Science, Ravello*, 1995.
61. T. Genet. Termination proofs using gpo ordering constraints. In M. Bidoit and M. Dauchet, editors, *Proceedings 22nd International Colloquium on Trees in Algebra and Programming*, volume 1214 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1997.
62. T. Genet. Proving termination of sequential reduction relation using tree automata. Manuscript; INRIA Lorraine, 1997.
63. Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proc. POPL'90*, pages 81–94. ACM, 1990.
64. F. Giunchiglia, C. Pecchiari, and C. Talcott. Reasonig theories: towards an architecture for open mechanized reasoning systems. Technical Report 9409-15, IRST, University of Trento, November 1994. Also in *Workshop on Frontiers of Combining Systems, FROCOS'96, 1996*.
65. J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
66. J. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Diaz and I. Ramos, editors, *Formalizing Programming Concepts*, pages 292–309. Springer-Verlag, 1981. LNCS, Volume 107.

67. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1992. To appear in J.A. Goguen and G.R. Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press, 1998.
68. C. Gunter and V. Gehlot. Nets as tensor theories. Technical Report MS-CIS-89-68, Dept. of Computer and Information Science, University of Pennsylvania, 1989.
69. T. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J. de Bakker, G. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, 1991.
70. H. Ishikawa, J. Meseguer, T. Watanabe, K. Futatsugi, and H. Nakashima. On the semantics of GAEA—an object-oriented specification of a concurrent reflective language in rewriting logic. In *IMSA'97*, pages 70–109. Information-Technology Promotion Agency, Japan, 1997.
71. M. Ishisone and T. Sawada. Brute: brute force rewriting engine. In *in Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.
72. ISO. *IS8807 : Information Processing Systems - Open System Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behavior*. ISO, February 1989.
73. ITU-T. *Recommendation X.903 — ISO/IEC International Standard 10746-3: "ODP Reference Model: Prescriptive Model"*. ISO, 1995.
74. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1993.
75. R. Jagannathan. Dataflow models. In E. Zoyama, editor, *Parallel and Distributed Computing Handbook*, pages 223–238. McGraw Hill, 1996.
76. N. Jones. Mix ten year later. In *PEPM'95*, pages 24–38. ACM-SIGPLAN, 1995.
77. R. Junglclaus, G. Saake, T. Hartmann, and C. Sernadas, editors. *Object-oriented specification of information systems: the TROLL language*. Technische Universität Braunschweig, Information-Berichte 91-04, 1991.
78. G. Kelly and R. Street. Review of the elements of 2-categories. In G. Kelly, editor, *Category Seminar, Sydney 1972/73*, pages 75–103. Springer Lecture Notes in Mathematics No. 420, 1974.
79. C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In V. Saraswat and P. van Hentryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 133–160. MIT Press, 1995.
80. C. Kirchner and P. Viry. Implementing parallel rewriting. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, pages 123–138. Springer LNAI 590, 1992.
81. H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proc. Rewriting Techniques and Applications, Kaiserslautern*, 1995.
82. H. Kirchner and P.-E. Moreau. Computational reflection and extension in ELAN. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
83. A. Knapp. Case studies with CafeOBJ. In *in Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.

84. P. Kosiuczenko and M. Wirsing. Timed rewriting logic, 1995. Working material for the 1995 Marktoberdorf International Summer School "Logic of Computation".
85. S. Krogdahl and O. Lysne. Verifying a distributed list system: a case history. *Formal Aspects of Computing*, 9:98–118, 1997.
86. C. Landauer. Discrete event systems in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
87. C. Laneve and U. Montanari. Axiomatizing permutation equivalence in the λ-calculus. In H. Kirchner and G. Levi, editors, *Proc. Third Int. Conf. on Algebraic and Logic Programming, Volterra, Italy, September 1992*, volume 632 of *LNCS*, pages 350–363. Springer-Verlag, 1992.
88. C. Laneve and U. Montanari. Axiomatizing permutation equivalence. *Mathematical Structures in Computer Science*, 1994. To appear.
89. F. W. Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences*, 50:869–873, 1963. Summary of Ph.D. Thesis, Columbia University.
90. U. Lechner. Object-oriented specification of distributed systems in the μ-calculus and Maude. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
91. U. Lechner. *Object-oriented specification of distributed systems.* PhD thesis, University of Passau, 1997.
92. U. Lechner and C. Lengauer. Modal μ-Maude. In *Object Orientation with Parallelism and Persistence*, B. Freitag, C.B. Jones, C. Lengauer and H.-J. Schek, editors, Kluwer, 1996.
93. U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. How to overcome the inheritance anomaly. in *Proc.ECOOP'96*, Springer LNCS, 1996.
94. U. Lechner, C. Lengauer, and M. Wirsing. An object-oriented airport. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, Santa Margherita, Italy, May/June 1994*, pages 351–367. Springer LNCS 906, 1995.
95. J. Levy. A higher order unification algorithm for bi-rewriting systems. In J. Agustí and P. García, editors, *Segundo Congreso Programación Declarativa*, pages 291–305, Blanes, Spain, September 1993. CSIC.
96. J. Levy and J. Agustí. Bi-rewriting, a term rewriting technique for monotonic order relations. In C. Kirchner, editor, *Proc. Fifth Int. Conf. on Rewriting Techniques and Applications, Montreal, Canada, June 1993*, volume 690 of *LNCS*, pages 17–31. Springer-Verlag, 1993.
97. J.-J. Lévy. Optimal reductions in the lambda calculus. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.
98. P. Lincoln, N. Martí-Oliet, and J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G. Blelloch, K. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.
99. P. Lincoln, N. Martí-Oliet, J. Meseguer, and L. Ricciulli. Compiling rewriting onto SIMD and MIMD/SIMD machines. In *Proceedings of PARLE'94, 6th In-*

ternational Conference on Parallel Architectures and Languages Europe, pages 37–48. Springer LNCS 817, 1994.
100. P. Lincoln, J. Meseguer, and L. Ricciulli. The Rewrite Rule Machine Node Architecture and its Performance. In *Proceedings of CONPAR'94, Linz, Austria, September 1994*, pages 509–520. Springer LNCS 854, 1994.
101. R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, April 1987.
102. S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
103. N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic. In D. P. et al., editor, *Category Theory and Computer Science*, pages 313–340. Springer LNCS 389, 1989. Final version in *Mathematical Structures in Computer Science*, 1:69-101, 1991.
104. N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic through categories: a survey. *Intl. J. of Foundations of Comp. Sci.*, 2(4):297–399, 1991.
105. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.
106. N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.
107. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
108. N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhoefer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*. 1998. To be published by Kluwer Academic Publishers.
109. I. Mason and C. Talcott. A semantics preserving actor translation. In *Proc. ICALP'97*, pages 369–378. Springer LNCS 1256, 1997.
110. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
111. J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.
112. J. Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.
113. J. Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of the Concur'90 Conference, Amsterdam, August 1990*, pages 384–400. Springer LNCS 458, 1990.
114. J. Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, February 1990. Revised June 1990.
115. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
116. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Di-*

rections in Concurrent Object-Oriented Programming, pages 314–390. MIT Press, 1993.
117. J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. M. Nierstrasz, editor, *Proc. ECOOP'93*, pages 220–246. Springer LNCS 707, 1993.
118. J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proc. CONCUR'96, Pisa, August 1996*, pages 331–372. Springer LNCS 1119, 1996.
119. J. Meseguer. Formal interoperability. In *Proceedings of the 1998 Conference on Mathematics in Artificial Intelligence, Fort Laurerdale, Florida, January 1998*, 1998. `http://rutcor.rutgers.edu/~amai/Proceedings.html`.
120. J. Meseguer. Membership algebra as a semantic framework for equational specification. in F. Parisi-Presicce, ed., Proc. WADT'97, Springer LNCS 1376, 1998.
121. J. Meseguer, K. Futatsugi, and T. Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. In *Proceedings of the 1992 International Symposium on New Models for Software Architecture, Tokyo, Japan, November 1992*, pages 61–106. Research Institute of Software Engineering, 1992.
122. J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.
123. J. Meseguer and U. Montanari. Mapping tile logic into rewriting logic. in F. Parisi-Presicce, ed., Proc. WADT'97, Springer LNCS 1376, 1998.
124. J. Meseguer and X. Qian. A logical semantics for object-oriented databases. In *Proc. International SIGMOD Conference on Management of Data*, pages 89–98. ACM, 1993.
125. J. Meseguer and C. Talcott. Reasoning theories and rewriting logic. Manuscript, Stanford University, June 1996.
126. J. Meseguer and C. Talcott. Using rewriting logic to interoperate architectural description languages (I and II). Lectures at the Santa Fe and Seattle DARPA-EDCS Workshops, March and July 1997. `http://www-formal.stanford.edu/clt/ArpaNsf/adl-interop.html`.
127. J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. L. Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.
128. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
129. S. Morasca, M. Pezzè, and M. Trubian. Timed high-level nets. *J. of Real-Time Systems*, 3:165–189, 1991.
130. P.-E. Moreau and H. Kirchner. Compilation techniques for associative-commutative normalisation. In *Proceedings of the International Workshop on Theory and Practice of Algebraic Specifications (Amsterdam, Holland)*, 1997.
131. P. Mosses. *Action Semantics*. Cambridge University Press, 1992.
132. E. Najm and J.-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems, Vol. 2*, pages 157–176. Chapman & Hall, 1997.
133. S. Nakajima. Encoding mobility in CafeOBJ: an exercise of describing mobile code-based software architecture. In *in Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.

134. S. Nakajima and K. Futatsugi. An object-oriented modeling method for algebraic specifications in CafeOBJ. In *Proceedings, 19th International Conference on Software Engineering*, pages 34–44. IEEE Computer Society Press, May 1997.
135. H. Nakashima. Organic programming for situation-thick AI systems. In *IMSA'97*, pages 156–163. Information-Technology Promotion Agency, Japan, 1997.
136. K. Ogata and K. Futatsugi. An abstract machine for order-sorted conditional term rewriting systems. In H. Comon, editor, *Proceedings of the 8th Conference on Rewriting Techniques and Applications*. Springer LNCS 1232, 1997.
137. K. Ohmaki, K. Futatsugi, and K. Takahashi. A basic LOTOS simulator in OBJ. In *Proceedings of the International Conference of Information Technology Commemorating the 30th Anniversary of the Information Processing Society of Japan (InfoJapan'90)*, pages 497–504. IPSJ, October 1990.
138. P. C. Ölveczky, P. Kosiuczenko, and M. Wirsing. An object-oriented algebraic steam-boiler control specification. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *The Steam-Boiler Case Study Book*. Springer-Verlag, 1996. To appear.
139. P. C. Ölveczky and J. Meseguer. Specifying real-time systems in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
140. I. Pita and N. Martí-Oliet. A Maude specification of an object oriented database model for telecomunication networks. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
141. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.
142. J.-C. Raoult and F. Voisin. Set-theoretic graph rewriting. In H.-J. Schneider and H. Ehrig, editors, *Graph Transformations in Computer Science*, pages 312–325. Springer LNCS 776, 1994.
143. H. Reichel. An approach to object semantics based on terminal co-algebras. To appear in *Mathematical Structures in Computer Science*, 1995. Presented at *Dagstuhl Seminar on Specification and Semantics*, Schloss Dagstuhl, Germany, May 1993.
144. W. Reisig. *Petri Nets*. Springer-Verlag, 1985.
145. C. Ringeissen. Prototyping combination of unification algorithms with the ELAN rule-based programming language. In H. Comon, editor, *Proceedings of the 8th Conference on Rewriting Techniques and Applications*. Springer LNCS 1232, 1997.
146. G. Saake and A. Sernadas, editors. *Information Systems—Correctness and Reusability*. Technische Universität Braunschweig, Information-Berichte 91-03, 1991.
147. H.-J. Schneider and H. Ehrig, editors. *Graph Transformations in Computer Science*. Springer LNCS 776, 1994.
148. M. Schorlemmer. Bi-rewriting rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
149. M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, 1996.
150. M.-O. Stehr. A rewriting semantics for algebraic Petri nets. Manuscript, March 1998, SRI International and C.S. Dept., Univ. of Hamburg, 1998.

151. J. Stell. Modelling term rewriting systems by sesqui-categories. Technical Report TR94-02, Keele University, 1994. Also in shorter form in Proc. C.A.E.N., 1994, pp. 121–127.
152. C. L. Talcott. An actor rewrite theory. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
153. C. L. Talcott. Interaction semantics for components of distributed systems. In E. Najm and J.-B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 154–169. Chapman & Hall, 1997.
154. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
155. V. F. Turchin. *Refal-5: programming guide and reference manual*. New England Publishing Co., 1989.
156. W. van der Aalst. Interval timed coloured Petri nets and their analysis. In M. A. Marsan, editor, *Application and Theory of Petri Nets 1993*, pages 453–472. Springer LNCS 691, 1993.
157. A. van Deursen. *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.
158. P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *LNCS*, pages 648–660. Springer-Verlag, 1994.
159. P. Viry. Input/output for ELAN. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
160. M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes*. PhD thesis, Université Henry Poincaré — Nancy I, 1994.
161. M. Wand and D. Friedman. The mystery of the tower revealed. *Lisp and Symbolic Computation*, 1(1):11–38, 1988.
162. M. Wirsing and A. Knapp. A formal approach to object-oriented software enginering. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
163. M. Wirsing, F. Nickl, and U. Lechner. Concurrent object-oriented design specification in SPECTRUM. Technical report, Institut für Informatik, Universität München, 1995.

This article was processed using the LaTeX macro package with LLNCS style