Rewriting Logic as a Semantic Framework for Concurrency: a Progress Report*

José Meseguer

SRI International, Menlo Park, CA 94025

Abstract. This paper surveys the work of many researchers on rewriting logic since it was first introduced in 1990. The main emphasis is on the use of rewriting logic as a *semantic framework* for concurrency. The goal in this regard is to express as faithfully as possible a very wide range of concurrency models, each on its own terms, avoiding any encodings or translations. Bringing very different models under a common semantic framework makes easier to understand what different models have in common and how they differ, to find deep connections between them, and to reason across their different formalisms. It becomes also much easier to achieve in a rigorous way the *integration* and *interoperation* of different models and languages whose combination offers attractive advantages. The logic and model theory of rewriting logic are also summarized, a number of current research directions are surveyed, and some concluding remarks about future directions are made.

Table of Contents

* Supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9224005, and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

1 Introduction

Since the first conference paper on rewriting logic in CONCUR'90 [73], dozens of authors in Europe, the US, Japan, and Northern Africa have vigorously advanced the rewriting logic research program in over sixty papers. By a serendipitous coincidence, the first international workshop on rewriting logic will take place in Asilomar, California, the week after CONCUR'96. The time seems somehow ripe for taking stock of the advances that have taken place, and I am very grateful to the organizers of CONCUR'96 for having given me the opportunity and the stimulus to do so.

Although I will try to cover most of the work that I am aware of, my main emphasis in this talk will be on rewriting logic as a *semantic framework* for concurrency. Rewriting logic has many theories and many models. Therefore, the task of providing a semantics is not conceived at all as a search for a *universal model* of concurrency, into which other models can be translated. The goal is very different, namely, to express as faithfully as possible each model on its own terms, avoiding any encodings or translations. In fact, given the wide variety of concurrency models and of concurrency phenomena the search for a universal model seems futile.

Bringing very different models under a common semantic framework has important conceptual and practical advantages. Conceptually, it becomes easier to understand what different models have in common and how they differ, to find deep connections between them, and to reason across their different formalims. In practice, it becomes much easier to achieve in a rigorous way the *integration* and *interoperation* of different models and languages whose combination offers attractive advantages. Section 4 explains how different models of concurrent computation can be naturally represented in rewriting logic. Some of these model representations were known from earlier work [74, 75]; they are briefly reviewed here in an updated form. Others, such as the treatment of simultaneous access to objects, graph rewriting, dataflow, neural networks, and real-time systems are new and are therefore discussed in greater detail.

To place the subject in perspective, the paper begins with a review of rewriting logic and its model theory in Sections 2 and 3. Section 5 surveys the different rewriting logic language implementation efforts in Europe, the US and Japan, including both interpreters and parallel implementations. Section 6 surveys other developments and research directions and the progress made in them. Finally, Section 7 makes some remarks about future directions.

2 Rewriting Logic

A *signature* in (order-sorted) rewriting logic is an (order-sorted) equational theory (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 [31] 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

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

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

[2] 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 [75] 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.

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, 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*. As we shall see, such equations can have a dramatic impact on the amount of concurrency available in a system. They work as "data solvents" to loosen up the data structures so that more rewrites can be performed in parallel.

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. In our view these latter logics support a nonexecutable—as far as the system described is concerned—level of specification above that of rewriting logic. Narciso Martí-Oliet and I are currently studying how these two levels can be best integrated within a unified wide-spectrum approach to the specification, prototyping, and declarative programming of concurrent systems; Ulrike Lechner has independently proposed a two-level integration of this kind as well [56, 55].

3 Models

We first sketch the construction of initial and free models for a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$. Such models capture nicely the intuitive idea of a "rewrite system" in the sense that they are systems whose states are E-equivalence classes of terms, and whose transitions are concurrent rewritings using the rules in R. 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:[3]

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.

[3] 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 to such descriptions specify the same abstract computation. We will see in Section 4.4 that for Petri nets this notion of true concurrency coincides with the well-known notion of commutative processes of a net.

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

Lemma 1. *[75] 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 2. 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[4] $r_{\mathcal{S}} * F = F^n * r_{\mathcal{S}'}$, 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 [75], where the soundness and completeness of rewriting logic for $\mathcal{R}$-system models is also proved.

Theorem 3. *$\mathcal{T}_{\mathcal{R}}$ 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 $F^{\natural} : \mathcal{T}_{\mathcal{R}}(X) \longrightarrow \mathcal{S}$.*

[4] Note that we use diagrammatic order for the *horizontal*, $\alpha * \beta$, and *vertical*, $\gamma;\delta$, composition of natural transformations [67].

4 Rewriting Logic as a Semantic Framework for Concurrency

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.

4.1 Parallel Functional Programming

Parallel functional programming is an important model of parallel computation. A lot of research has been devoted to parallel implementations of functional languages [96] and also to parallel dataflow and reduction architectures supporting them.

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.

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.

By contrast, the rewrite rules in a rewrite theory need nor be Church-Rosser, and may never terminate. Therefore, in general their concurrent execution cannot be understood as the computation of a unique functional value; often not even as the computation of *a* value at all. However, parallel 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 [68].

For first-order functional programs the above remarks make clear how they can be regarded as a special case of rewriting logic. However, for higher-order functions, since their formalization is somewhat different, more has to be said. 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 [75]. An illuminating investigation of parallel computations in the lambda calculus using rewriting logic has been carried out by Laneve and Montanari [52, 53].

Before addressing the case of the lambda calculus, Laneve and Montanari [53] first clarify the exact relationship between the equivalence of rewrites obtained by the equations identifying proof terms in the free model $\mathcal{T}_{\mathcal{R}}(X)$ of a rewrite theory $\mathcal{R}$, and Boudol's notion of permutation equivalence for term rewriting systems using the residual calculus [17]. The theory $\mathcal{R}$ is assumed to have its equational part E empty (syntactic rewriting) and to be such that the rewrite rules are left-linear (no repeated variable occurrences in lefthand sides or rules) and have nonvariable lefthand sides that contain all the variables in their corresponding righthand sides. They prove that both notions of equivalence coincide, and conclude that the equational description in $\mathcal{T}_{\mathcal{R}}(X)$ is considerably simpler than that provided by the residual calculus.

Laneve and Montanari then go on to consider the 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.

4.2 Labelled Transition Systems

A labelled 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.

Their poverty has two aspects. Firstly, they are 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 labelled 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.

4.3 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 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.

4.4 Petri Nets

Petri nets have a straightforward and very natural expression as rewrite theories. Their distributed states correspond to *markings*, that is, to finite multisets of basic constants called *places*. Algebraically they are axiomatized by an *associative* and *commutative* multiset union operation with *identity* the empty multiset and with constants the places. A transition t in a place-transition net is simply a labelled rewrite rule $t : M \longrightarrow M'$ between two multiset markings [75]. Therefore, we can view a net $\mathcal{N}$ as a rewrite theory $\mathcal{N}$ with the above algebraic axiomatization for its markings and with one rewrite rule per transition, so that firing of a transition exactly corresponds to rewriting modulo associativity, commutativity and identity with the corresponding rewrite rule.

In this way, the finite concurrent computations of a net $\mathcal{N}$ are formalized as arrows in the category $\mathcal{T}_{\mathcal{N}}$. Specifically, they exactly correspond to the *commutative processes* of $\mathcal{N}$ in the sense of Best and Devillers [11]. This result, showing that the equational theory of true concurrency provided by rewriting logic agrees with more traditional notions of true concurrency in the case of Petri nets, has been proved by Degano, Meseguer and Montanari [28, 29] using an earlier categorical model of Petri net computations denoted $\mathcal{T}[\mathcal{N}]$ [80] 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 [35, 36], this was pointed out in [75]. Applications of rewriting logic to Petri net algebraic specification have been developed by Battiston, Crespi, De Cindio, and Mauri [8], and also by Bettaz and Maouche [12, 13].

4.5 Gamma and the Chemical Abstract Machine

The Gamma language of Banâtre and Le Mètayer [7], and Berry and Boudol's *chemical abstract machine*, or *cham* [66, 10], 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*, because the associativity and commutativity axioms enjoyed by multiset union allow the different elements to "float" freely in the expression, so as to come into contact with each other at will. 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*. This generalizes the place/transition Petri net case by allowing structured elements in the multiset, instead of just atomic places, so that both Gamma and the *cham* could in some sense be regarded as high-level Petri net formalisms.

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

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

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, with each machine possibly extending the basic syntax by additional function symbols. The common syntax is typed, and can be expressed as the following order-sorted signature Ω:

sorts Molecule, Molecules, Solution .
subsorts Solution $<$ *Molecule* $<$ *Molecules .*
op $\lambda : \longrightarrow$ *Molecules .*
op $\_,\_ :$ *Molecules Molecules* $\longrightarrow$ *Molecules .*
op $\{\!|\_|\!\} :$ *Molecules* $\longrightarrow$ *Solution .* `*** membrane operator`
op $\_ \triangleleft \_ :$ *Molecule Solution* $\longrightarrow$ *Molecule .* `*** airlock operator`

We can describe a *cham* 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 [75] for some more discussion.

4.6 CCS, LOTOS and the π-Calculus

Kokichi Futatsugi, Timothy Winkler and I [78], and in a different later version Narciso Martí-Oliet and I [68], 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 [68]. 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 [38] and is discussed in Section 6.1.

LOTOS [42] is a specification language combining the two formalisms of algebraic data types and (an extension of) CCS. It is pointed out in [78] 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 [86]. 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 [102] 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 [102].

More recently, Viry [101] 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 [84]. However, as in the case of rewriting logic specifications of the lambda calculus discussed in Section 4.1, binding operators become an extra feature that should be accounted for. As for the lambda calculus, the answer given by Viry [102] resides in an equational theory of explicit substitution, so that expressions up to alpha-conversion can be regarded as equivalence classes.

4.7 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 subtype declaration

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

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

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.

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

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

The concurrent behavior of bounded buffers that interact with other objects by `put` and `get` messages can then be axiomatized by the rewrite rules

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

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

Then, a configuration such as

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

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

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

by applying concurrently the two rewrite rules[5] for `put` and `get` modulo associativity and commutativity.

Intuitively, we can think of messages as "travelling" to come into contact with the objects to which they are sent and then causing "communication events" by application of rewrite rules. In rewriting logic, this travelling is accounted for in a very abstract way by the structural laws of associativity, commutativity, and identity. The above two rules illustrate the *asynchronous message passing* communication between objects typical of *Actor systems* [3, 2]. Generalizing slightly the Actor case, the Simple Maude language [62] adopts the following general form of conditional rules for asynchronous message passing interaction between objects

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

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 $(\langle O : F' \mid \mathit{atts}' \rangle)$ means that the object O—in a possibly different state—is only an optional part of the righthand side, i.e., that it can be omitted in some rules so that the object is then deleted. In addition, p new objects may be created, and q new messages may be generated for $p, q \geq 0$.

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

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

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

The above representation of objects is the one adopted in the Maude language [72, 83, 76]. It assumes, as it is common in many object-oriented systems, that each object has a unique name, different from that of all other objects. In

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

fact, by giving appropriate rewrite rules for the creation and deletion of objects [76], it is not hard to ensure that this property is preserved by the rules of an object-oriented system. However, rewriting logic as such is neutral about the treatment of object identity and many other such matters. For example, for purposes of grouping objects in *components* that can be combined with each other to form bigger open systems, Carolyn Talcott adopts instead a more abstract representation of objects as *abstract actors* [100], where objects can be renamed by a form of alpha-conversion to avoid name clashes across components. In two very fine papers [100, 99] she then uses rewriting logic to reason formally about the behavior of actors, including their infinite fair computations.

An important problem in concurrent object-oriented programming to which rewriting logic has been successfully applied is the so-called *inheritance anomaly* [71], that is, the serious difficulties often encountered when trying to integrate object-oriented inheritance and concurrency in a programming language. The problem is that an object such as the bounded buffer described above may not be ready to process certain messages, such as a `put` when the buffer is full or a `get` when it is empty. The more or less ad-hoc solutions adopted to deal with this typically do not survive well the passage to subclasses in which more attributes and more methods may have been introduced. Using an order-sorted type structure [40], class inheritance can be naturally supported in rewriting logic; and message redefinition in subclasses can be described by appropriate composition operations between rewrite theories [76]. I used these ideas in [77] to show 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 [57] proposes additional rewriting logic techniques to give a somewhat different solution to this problem.

Object interaction need not be asynchronous. In systems different from Actor systems 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

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

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 for which efficient parallel implementation techniques exist. That is, in some particular cases the cost of synchronization can be quite low, and a direct implementation of the corresponding rules may be the most natural and efficient thing to do. However, in cases where the synchronization and communication demands become quite high, it may be better to consider such synchronous rules as higher level executable specifications of the desired behavior, and to implement them at a lower level by asynchronous message passing. The paper [62] shows that, under quite general assumptions, it is indeed possible to transform synchronous rules of the form (‡) into simpler Actor-like rules of the form (†).

Besides the work already cited, 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 [58]; and Wirsing Nickl and Lechner [106] have proposed the rewriting logic-based *OOSpectrum* formalism for formal object-oriented specifications. Wirsing has also been studying 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 [105]. From a different, (co-)algebraic, perspective, Reichel has found rewriting logic useful in his final coalgebra semantics for objects [91]. The benefits of rewriting logic for execution of, and formal reasoning about, object-oriented *discrete event simulations* is another application area currently being investigated by Landauer [51].

Another area where rewriting logic has proved useful is in the specification and programming of *object-oriented databases*. Meseguer and Quian [81] 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 [30] 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 [92, 44]. More recently, Pita and Martí-Oliet [89] have carried out a thorough case study on the application of Maude to the executable specification of a database model for broadcast telecommunication networks.

4.8 Unity

UNITY [19] 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 [75] 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 [75], but the basic idea is straightforward. In essence a UNITY program is a 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 expression $exp_i(x_1, \ldots, x_n)$ 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:

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

4.9 Concurrent Access to Objects

In an object system there are two kinds of concurrency, *inter-object* concurrency, thanks to the concurrent execution of rules of the form (†) or (‡), and *intra-object* concurrency. Intra-object concurrency is due to the fact that many rewrites can simultaneously update different parts of an object's internal state; this can typically be due to the concurrency of the different functional data types used in the attributes of an object.

Inter- and intra-object concurrency can be combined to maximize overall concurrency. This can be accomplished through a number of subobjects under an object, to which many tasks of the object can be delegated in parallel. For Maude, this idea of "objects within objects" was suggested in [76] and is also used in [57]. However, even if the message-processing computations of a master object are made very "lightweight" by the object delegating tasks to its subobjects, the fact remains that in the formalization proposed so far *the same object cannot be shared by two simultaneous rewrites*, and therefore such a master object may still become a bottleneck. Therefore, as things stand rewrites involving the same object must be sequentialized, even if by safely sharing the object concurrency would be increased. For example, a flight object in an object-oriented airline reservation database should be accessible by several transaction messages simultaneously. This can be done using the general idea that the equational axioms of a rewrite theory have as their purpose to "loosen" the distributed state of the system in order to make it more concurrent. If we picture the concurrent multiset top structure of an object-oriented system as water in which the objects float, we can imagine each object as an insoluble blob of oil. What we need are "copying" and "emulsifying" axioms that can either duplicate or break the blob of oil into several pieces, so that the pieces can interact concurrently.

Consider first the case of simultaneous reads to the same object. We say that an object O occurs in a *read-only* way in a rewrite rule if the state of the object is the same on both sides of the rule. We would therefore like to share such an object in two simultaneous rewrites. This can be accomplished by adding the following two "copying" equational axioms to our specification[6]

$$\begin{array}{l}\langle O : C \mid atts\rangle = \{O : C \mid atts \mid 0\} \\ \{O : C \mid atts \mid n\} = \{O : C \mid atts \mid n+1\}\ [O : C \mid atts]\end{array}$$

and by transforming the original rules so that each read-only occurrence of an object $\langle O : C \mid atts\rangle$ in a rule is replaced by the sharable read-only variant $[O : C \mid atts]$.

The entities being shared by means of such "copying" axioms do not have to be objects in an object-oriented system. They could for example be places in a Petri net, or any kind of element in a distributed system having a multiset structure. The slightly more general formulation

$$\begin{array}{l}x = \{x \mid 0\} \\ \{x \mid n\} = \{x \mid n+1\}\ [x]\end{array}$$

where x ranges over elements in the multiset, allows this, and yields as a special case the place/transition version of the *contextual nets with positive contexts* model of computation with shared reads proposed by Montanari and Rossi [85]. Specifically, a contextual net transition t with preconditions $a_1, \ldots, a_n$, postconditions $b_1, \ldots, b_m$, and positive context $c_1, \ldots, c_k$ becomes a rewrite rule

$$t : a_1 \ldots a_n\, [c_1] \ldots [c_k] \longrightarrow b_1 \ldots b_m\, [c_1] \ldots [c_k]$$

More generally, one would like to allow simultaneous reads and writes on the same object. This can be accomplished by making the notion of reading or writing local to particular attributes within an object. This is related to the notational convention in object-oriented Maude specifications [83, 76] of not mentioning in a given rule those attributes of an object that are not relevant for that rule. Indeed, let $\overline{a : v}$ denote the attribute-value pairs $a_1 : v_1, \ldots, a_n : v_n$, where the $\overline{a}$ are the attribute identifiers of a given class C having $\overline{s}$ as the corresponding sorts of values prescribed for those attributes. In this context, the v_i can be either terms (with or without variables) or variables of sort s_i. We allow rules where the attributes appearing in the lefthand and righthand side patterns for an object O mentioned in the rule need not exhaust all the object's attributes, but can instead be in any two arbitrary subsets of the object's attributes[7]. We can picture this as follows

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

[6] Where the counting of the read-only copies and their read-only use guarantee coherence, since all the copies must have been "folded back together" in order for the original object to be engaged in a rewrite that changes some of its attributes.

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

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

$$\begin{array}{l} \ldots\langle O : C \mid \overline{al : vl}, \overline{ab : vb}, \overline{ar : x}, atts\rangle \ldots \\ \longrightarrow \ldots\langle O : C \mid \overline{al : vl}, \overline{ab : vb'}, \overline{ar : vr}, atts\rangle \ldots \end{array}$$

where the $\overline{x}$ are new "don't care" variables and *atts* matches the remaining attribute-value pairs. The attributes mentioned only on the left are preserved unchanged, the original values of attributes mentioned only on the right don't matter, and all attributes not explicitly mentioned are left unchanged. Therefore, $\overline{al}$ are the *read-only* attributes actually involved in the rewrite, $\overline{ab}$ and $\overline{ar}$ are the *read* attributes, and the remaining attributes are not involved at all.

What we desire is a program transformation that replaces each rule

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

by the rule

$$\begin{array}{l} \ldots[O : C \mid \overline{al : vl}]\ \{O : C \mid \overline{ab : vb}, \overline{ar : x}\} \ldots \\ \longrightarrow \ldots[O : C \mid \overline{al : vl}]\ \{O : C \mid \overline{ab : vb'}, \overline{ar : vr}\} \ldots \end{array}$$

where $[O : C \mid \overline{al : vl}]$ is the sharable read-only part of the object, and $\{O : C \mid \overline{ab : vb}, \overline{ar : x}\}$ cannot be shared, and where possible additional attributes not involved in the rewrite are are again left implicit by convention.

With adequate equational axioms this allows several rules to simultaneously and coherently access the attributes of an object so that read-only attributes can take part in several simultaneous reads, but write attributes can only be modified by a single rewrite. The axioms in question achieve both "copying" and "emulsifying" effects and are quite simple

$$\begin{array}{l} \langle O : C \mid atts, att's\rangle = \{O : C \mid atts \mid att's \mid 0\} \\ \{O : C \mid atts \mid att's \mid n\} = \{O : C \mid atts \mid att's \mid n+1\}\ [O : C \mid atts] \\ \{O : C \mid atts \mid att's, att''s \mid n\} = \{O : C \mid atts \mid att''s \mid n+1\}\ \{O : C \mid att's\} \end{array}$$

What these axioms provide is a precise algebraic specification of what correct simultaneous access to objects should be, so that the concurrency of the system can be increased. They of course are like "magic," in that they give a declarative specification but do not prescribe any particular mechanism. A concrete implementation using for example locks on attributes, or concurrent aggregates [20], can then be judged correct relative to such a specification.

The above axioms are one way of illustrating how algebraic laws can axiomatize coherent simultaneous access to elements of a distributed state. The general method implicit in these examples is to increase the concurrency available in a rewrite theory $\mathcal{R}$ by defining a refinement map $\mathcal{R} \longrightarrow \mathcal{R}^\flat$, where we have added in $\mathcal{R}^\flat$ the additional equational laws that increase concurrency. Section 4.11 will present even simpler axioms for increasing concurrency in dataflow networks. It would be interesting to investigate equational laws for increased concurrency in areas such as coherent distributed memory models, and concurrent database transactions.

4.10 Graph Rewriting

Parallel graph rewriting is a model of computation of great importance. On the one hand, efficient implementations of functional languages often represent expressions as directed acyclic graphs rather than as trees, so that at the implementation level the rewriting taking place is graph rewriting. This case is usually called *term graph rewriting* and has been studied extensively (see [97] for a representative collection of papers). However, graph rewriting is a very general model and can express many other computations besides functional ones. The theory of graph grammars and graph transformations (see [33, 94] for recent conferences) considers graph rewriting in this more general sense.

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 [33, 94]. However, for our purposes the most convenient axiomatizations are those in which labelled 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 [9], Corradini and Montanari [26], and Raoult and Voisin [90].

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 [90] where graph rewriting is also understood as multiset rewriting. The basic idea is to consider each node of a labelled graph as an *object* with two attributes, one the data element labelling the node—which can belong to any desired data type of values—and the other a *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

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

An object with this information is essentially what is called a *hyperedge* in the terminology of graph grammars [27], 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 labelled 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 [62], 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.

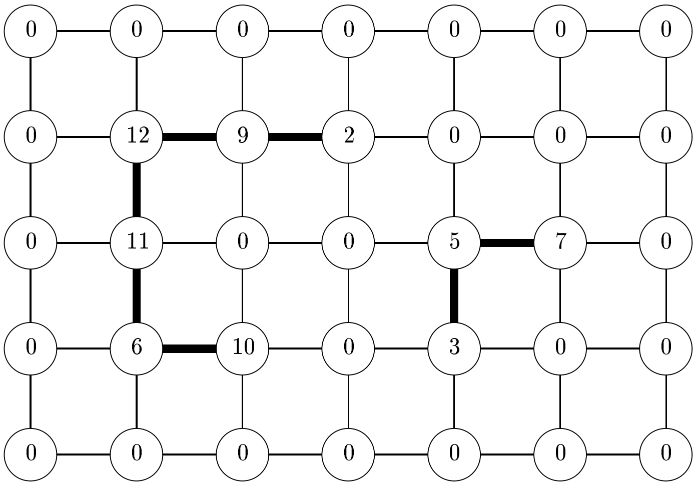

Fig. 1. Image as two-dimensional array.

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.

Note that only the value in node a may change as a result of applying this rule. Therefore, the remaining nodes act as "read-only" nodes that can be simultaneously read by other matches of the same rule, thus increasing concurrency. The rewriting logic expression of this rule, making the read-only nodes available for other matches, is simply,

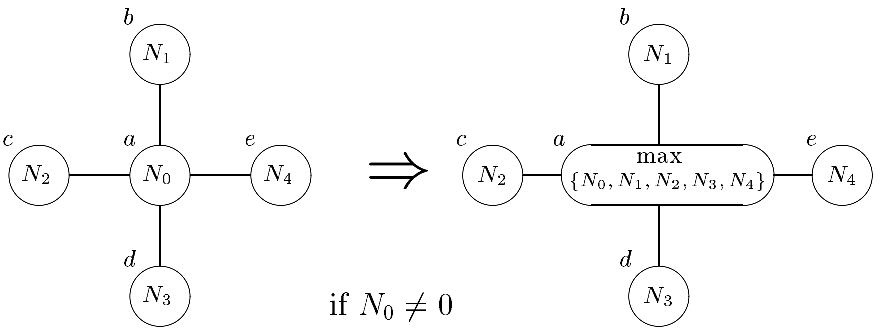

Fig. 2. A graph rewrite rule.

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

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

In this rule, as it happens, the *topology* of the graph does not change. As explained in [62, 63], rules with this additional property, since they allow efficient data mapping at compile time and do not require any garbage collection, can be implemented in parallel very efficiently. They typically express highly regular parallel computing problems.

In general, however, a graph rewrite rule may change the links between the nodes and may add new nodes, but it is not allowed to remove nodes. In addition, it should not create "dangling pointers," which is guaranteed if the new links added in the righthand side all correspond to nodes appearing in the rule. As usual for object-oriented systems, the rule must also satisfy the additional requirement of never leading to a situation where two objects have the same name. Several simple solutions in the style described in [76] are possible for this. For example, we may assume that, when rules can create new nodes, the values stored in the nodes are actually pairs (v, n) consisting of the actual value v and a natural number n. Then, one of the "write" objects, say A, appearing on the lefthand side with value (v, n) can be used to create k fresh new names $A.n.1, \ldots, A.n.k$ for the k new nodes appearing on the righthand side, where the value of A now becomes $(v', n + 1)$ for an appropriate v'. This will always work, provided that the node names in the initial graph being rewritten are all different and do not involve strings of numbers.

The requirement that no nodes are deleted might seem restrictive, but in practice it isn't. It has the great advantage of making the matching of a graph rewrite rule completely *local* and *context independent*, which it is not the case in either the pushout constructions, or in the formulation of Raoult and Voisin [90]. By contrast, a context-dependent rule in which one needs to check that no dangling pointers are created in the rest of the graph by the elimination of a node may require a *global search* in an arbitrarily big context graph before each rewrite step.

Of course garbage can be produced, but it can be collected with standard techniques. For example, for acyclic graphs it is enough to add a reference counter as a new attribute in a sublclass *RcNode* with objects of the form

$$\langle a : \mathit{RcNode} \mid \mathit{val} : v, \mathit{rc} : n, \mathit{links} : l \rangle$$

and to update the reference count of those nodes in a rule's righthand side whose references have been modified by the rule. Assuming that nodes of the graph with zero reference count become garbage, one can then collect such garbage with rules

$$\begin{array}{l} \langle A : \mathit{RcNode} \mid \mathit{rc} : 0, \mathit{links} : B\ L \rangle\ \langle B : \mathit{RcNode} \mid \mathit{rc} : n \rangle \\ \longrightarrow \langle A : \mathit{RcNode} \mid \mathit{rc} : 0, \mathit{links} : L \rangle\ \langle B : \mathit{RcNode} \mid \mathit{rc} : n - 1 \rangle \end{array}$$

$$\langle A : \mathit{RcNode} \mid \mathit{rc} : 0, \mathit{links} : \mathit{nil} \rangle \longrightarrow \mathit{null}$$

The case of cyclic graphs could be handled with other techniques such as mark-and-sweep that can also be specificed by adequate rules.

4.11 Dataflow

The dataflow model of computation [43] has been thoroughly investigated both at the level of parallel functional languages, and in terms of parallel architectures directly supporting the dataflow model. In fact, it is one of the contending models of parallel computing and parallel architecture. The model is very intuitive. One pictures the computation as a graph in which data flows along edges and is computed in nodes labelled 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 integers, the reals, or some 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, down to the case of holding a single value—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 $\langle e : \mathit{Edge} \mid \mathit{cts} : c \rangle$ 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. Typically it satisfies the restriction that each edge links exactly two nodes; duplication of data is made explicit by duplicator nodes. However, this restriction could be relaxed. 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. Intuitively, however, the same edge could simultaneously receive the output of one rewrite and provide one of the inputs to another, so as to maximize the concurrency of the net. The equational axiom allowing simultaneous input and output access to an edge in the piped model is very simple, namely,

$$\langle e : Edge \mid cts : l\, x \rangle = [e : Edge \mid cts : l]_{in}\; [e : Edge \mid cts : x]_{out}$$

The firing rule for a functional node computing a function f of n arguments in the piped model is then

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

where we of course assume that the equations in our rewrite theory axiomatize the data algebra, so as to compute the actual value of $f(x_1 \ldots x_n)$. Similarly, the behavior of a T-gate control node that lets an element from its second input go through when the first input is true, and discards that same element when it is false, is axiomatized by the rules

$$\begin{array}{l} [e_1 : Edge \mid cts : true]_{out}\; [e_2 : Edge \mid cts : x]_{out} \\ \langle a : TGate \mid inputs : e_1\, e_2, output : e \rangle \\ [e : Edge \mid cts : l]_{in} \\ \qquad \longrightarrow [e_1 : Edge \mid cts : nil]_{out}\; [e_2 : Edge \mid cts : nil]_{out} \\ \qquad\qquad \langle a : TGate \mid inputs : e_1\, e_2, output : e \rangle \\ \qquad\qquad [e : Edge \mid cts : x\, l]_{in} \end{array}$$

$$\begin{array}{l} [e_1 : Edge \mid cts : false]_{out}\; [e_2 : Edge \mid cts : x]_{out} \\ \langle a : TGate \mid inputs : e_1\, e_2, output : e \rangle \\ [e : Edge \mid cts : l]_{in} \\ \qquad \longrightarrow [e_1 : Edge \mid cts : nil]_{out}\; [e_2 : Edge \mid cts : nil]_{out} \\ \qquad\qquad \langle a : TGate \mid inputs : e_1\, e_2, output : e \rangle \\ \qquad\qquad [e : Edge \mid cts : l]_{in} \end{array}$$

The firing rule for a duplicator node ($\triangle$) is also straightforward, namely,

$$\begin{array}{l}[e : \mathit{Edge} \mid \mathit{cts} : x]_{out} \\ \langle a : \triangle \mid \mathit{input} : e, \mathit{outputs} : e_1\, e_2 \rangle \\ [e_1 : \mathit{Edge} \mid \mathit{cts} : l_1]_{in}\ [e_2 : \mathit{Edge} \mid \mathit{cts} : l_2]_{in} \\ \qquad \longrightarrow [e : \mathit{Edge} \mid \mathit{cts} : \mathit{nil}]_{out} \\ \qquad\quad \langle a : \triangle \mid \mathit{input} : e, \mathit{outputs} : e_1\, e_2 \rangle \\ \qquad\quad [e_1 : \mathit{Edge} \mid \mathit{cts} : x\, l_1]_{in}\ [e_2 : \mathit{Edge} \mid \mathit{cts} : x\, l_2]_{in}\end{array}$$

A dataflow network may be embedded in a highly unpredictable environment, such as for example a collection of sensors providing streams of inputs to the network. Such an environment can also be formalized by input nodes and a nondeterministic rewrite rule

$$\begin{array}{l}\langle a : \mathit{Input} \mid \mathit{output} : e \rangle\ [e : \mathit{Edge} \mid \mathit{cts} : l]_{in} \\ \longrightarrow \langle a : \mathit{Input} \mid \mathit{output} : e \rangle\ [e : \mathit{Edge} \mid \mathit{cts} : x\, l]_{in}\end{array}$$

where x is a variable of appropriate type.

Variants in which the FIFO buffer is bounded, as well as the tagged case can be handled with similar firing rules.

4.12 Neural Networks

Artificial neural networks [65] are another important model of parallel computation. They are particularly well-suited for providing massively parallel solutions to pattern recognition problems. They have been around since the early days of cybernetics, but have received renewed attention in recent years due to the many practical applications that have been found for them. The basic idea behind artificial neural networks 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. All such firings may happen asynchronously, thus yielding an intrinsically parallel model of computation.

The formalization in rewriting logic is very direct. Neurons can be regarded as objects

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

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,

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

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. To allow as much parallelism as possible in a neural network, we should allow all the inputs of a neuron to be received asynchronously and even concurrently, and the output to be similarly received by their targets. This can be accomplished using the techniques for simultaneous access to objects described in Section 4.9 by means of the single rewrite rule

$$
\begin{aligned}
&\{a : \mathit{Neuron} \mid \mathit{out}(b) : \mathit{false}\}\ [a : \mathit{Neuron} \mid \mathit{output}(y)]\\
&\{b : \mathit{Neuron} \mid \mathit{in}(a) : (w, mt)\}\\
&\qquad \longrightarrow \{a : \mathit{Neuron} \mid \mathit{out}(b) : \mathit{true}\}\ [a : \mathit{Neuron} \mid \mathit{output}(y)]\\
&\qquad\quad\ \ \{b : \mathit{Neuron} \mid \mathit{in}(a) : (w, y)\}
\end{aligned}
$$

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. Also, as in the dataflow case, the inputs from an external environment can be formalized by means of nondeterministic rewrite rules involving input nodes.

4.13 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) [50], 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 [87]. In fact, rewriting logic object-oriented specifications in the Maude language have a natural extension to TRL object-oriented specifications in *Timed Maude* [87].

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 are currently investigating a different alternative, namely, the suitability of standard rewriting logic for directly specifying real-time systems. This seems an interesting research problem in its own right, and has the advantage of making available for real-time specification the different language tools that have been developed for standard rewriting logic.

A number of frequently used models of real-time computation have a very natural and direct expression in standard rewriting logic. One of the models that Ölveczky and I consider in [88] is *timed automata*. Omitting details about initial states and acceptance conditions, a timed automaton (see, e. g., [6]) consists of

- a finite alphabet Σ,
- a finite set S of states,
- a finite set C of clocks,
- a set $\Phi(C)$ of clock constraints, and
- a set $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ of transitions. The tuple $\langle s, s', a, \lambda, \phi \rangle$ represents a transition from state s to state s' on input symbol a. The set $\lambda \subseteq C$ gives the clocks to be reset with this transition, and ϕ is a clock constraint over C.

Given a timed word (i. e. a sequence of tuples $\langle a_i, r_i \rangle$ where a_i is an input symbol and r_i is the time at which it occurs), the automaton starts at time 0 with all clocks initialized to 0. As time advances the values of all clocks change, reflecting the elapsed time; that is, the state of the automaton can change not only by the above transitions, but also by the passage of time, with all the clocks being increased by the same amount. At time r_i the automaton changes state from s to s' using some transition of the form $\langle s, s', a_i, \lambda, \phi \rangle$ reading input a_i, if the current values of the clocks satisfy ϕ. With this transition the clocks in λ are reset to 0, and thus start counting time again.

A timed automaton can be naturally represented in rewriting logic. The time domain and its associated constraints $\Phi(C)$ are equationally axiomatized. Then, the tuple $\langle s, c_1, \ldots, c_n \rangle$ represents an automaton in state s such that the values of the clocks in C are $c_1, \ldots, c_n$. Each transition $\langle s, s', a, \lambda, \phi \rangle$ is then expressed as a rewrite rule

$$a : \langle s, c_1, \ldots, c_n \rangle \longrightarrow \langle s', c'_1, \ldots, c'_n \rangle \ \textit{if} \ \phi(c_1, \ldots, c_n)$$

where $c'_i = 0$ if $c_i \in \lambda$, and $c'_i = c_i$ otherwise. In addition, a rewrite rule

$$\textit{tick} : \langle x, c_1, \ldots, c_n \rangle \longrightarrow \langle x, c_1 + r, \ldots, c_n + r \rangle$$

(with r a variable in the time domain) is added to represent the elapse of time. Other real-time models such as *timed transition systems* [41] and *hybrid automata* [5, 4] have similar straightforward formulations within standard rewriting logic [88]. Ölveczky and I are also investigating a translation of TRL into rewriting logic with the purpose of making Timed Maude executable using the Maude interpreter.

5 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 parallel programming in rewriting logic.

5.1 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 6.4, 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 developed at INRIA Lorraine by Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau and Vittek [46, 103, 16] has as modules *computational systems*, consisting of a rewrite theory and a *strategy* to guide the rewriting process. 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 [46, 103, 49, 14, 18]. 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 [101].

Futatsugi and Sawada at Japan Advanced Institute of Science and Technology (JAIST) in Kanazawa are in a very mature stage of development of their Cafe language [37], which is also based on rewriting logic, contains OBJ as its functional sublanguage, and supports object-oriented specifications. The Cafe language will be the basis of an ambitious research effort 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.

In our group at SRI, Manuel Clavel, Steven Eker, Patrick Lincoln and I are working on the implementation of an interpreter for Maude [83, 76, 22]. Maude is based on a typed version of rewriting logic that is order-sorted and supports *sort constraints* [79]. It has *functional modules*, that are Church-Rosser and terminating equational theories, *system modules*, that specify general rewrite theories, and *object-oriented modules*, that provide syntactic sugar for object-oriented rewrite theories. These modules can be combined by module composition operations in the OBJ style. Its rewrite engine 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 [34]. In addition, Maude supports reflective rewriting logic computations, and has flexible evaluation strategies.

5.2 Parallel Programming 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 parallel programming. We have therefore 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, in the sense that Maude modules are rewriting logic theories in that subset, and can be supported by an interpreter implementation adequate for rapid prototyping, debugging, and executable specification. The second, smaller subset gives rise to Simple Maude [62], a sublanguage meant to 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. Figure 3 summarizes the three levels involved.

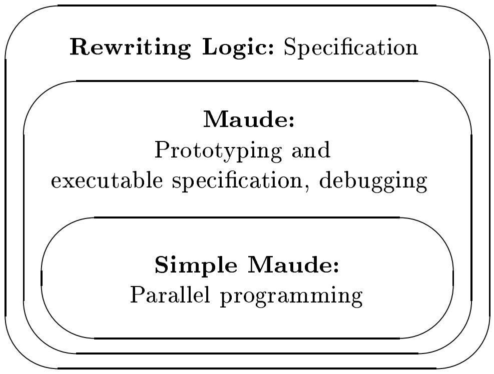

Fig. 3. Maude and Simple Maude as subsets of rewriting logic.

In Simple Maude, three types of rewriting, all of which can be efficiently implemented, are supported. Together, they cover a very wide variety of applications; they are:

Term rewriting. In this case, the data structures being rewritten are *terms*, that is, syntactic expressions that can be represented as labelled trees or acyclic graphs. Functional programming falls within this type of rewriting, that does also support nonconfluent term rewrite rules, and rewriting modulo confluent and terminating structural axioms E. Symbolic computations are naturally expressible using term rewrite rules.

Graph rewriting. In this case, the data structures being rewritten are *labelled graphs*. For general graph rewrite rules, the graph can evolve by rewriting in highly unpredictable ways. A very important subcase is that of graph rewrite rules for which the *topology* of the data graph remains unchanged after rewriting. Many highly regular computations, including many scientific computing applications, cellular automata algorithms, and systolic algorithms, fall within this fixed-topology subclass, for which adequate placement of the data graph on a parallel machine can lead to implementations with highly predictable and often quite low communications costs.

Object-oriented rewriting. This case corresponds to actor-like objects that interact with each other by asynchronous message-passing. Abstractly, the distributed state of a concurrent object-oriented system of this kind can be naturally regarded as a *multiset* data structure made up of objects and messages; the concurrent execution of messages then corresponds to concurrently rewriting this multiset by means of appropriate rewrite rules. In a parallel machine this is implemented by *communication* on a network, on which messages travel to reach their destination objects. Many applications are naturally expressible as concurrent systems of interacting objects. For example, many discrete event simulations, and many distributed AI and database applications can be naturally expressed and parallelized in this way.

The design of Simple Maude seeks to support a very wide range of parallel programming applications in an efficient and natural manner. In this sense Simple Maude is a *multiparadigm* parallel programming language that includes a *functional* facet (the term rewriting case), a *concurrent object-oriented* facet (the object-oriented rewriting case), and a facet supporting *highly regular in-place computations* (the graph rewriting case). By supporting programming and efficient execution of each application in the facet better suited for it, the inadequacies—both in terms of expressiveness and efficiency—of a single-facet language are avoided, while the benefits of each facet are preserved in their entirety.

Patrick Lincoln, Livio Ricciulli and I have developed parallel compilation techniquies and a prototype Simple Maude compiler [63] that generates parallel code for the Rewrite Rule Machine that we are developing at SRI [64]. Our detailed simulation and performance studies with an RRM having 64 SIMD nodes indicate that very high performance—typically with two to three orders of magnitude speedups over the best sequential implementations on advanced workstations for the same problem—can be achieved with this approach to declarative parallel computing with rewrite rules.

At INRIA Lorraine, Viry and C. Kirchner [47] 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 case called term rewriting in the above classification, 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 objec-oriented rewrite theories [21]. 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.

6 Other Developments and Research Directions

I include here several research developments that are more topical and can give the reader a more comprehensive view of the entire research program.

6.1 2-Category Models

Lawvere [54] 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 $\hat{A} : \mathcal{L}_T \longrightarrow \mathbf{Set}$, when we view $[t(x_1, \ldots, x_n)]$ as an arrow $[t(x_1, \ldots, x_n)] : n \longrightarrow 1$ and compose such arrows by substitution. That is,

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

yields as a composition $[t(\overline{u}/\overline{x})] : m \longrightarrow 1$. In fact, 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 morphims 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 [74], 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 [67, 45] **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 a 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

$$
\begin{tikzcd}
m \arrow[r, bend left=50, "{([u_1],\ldots,[u_n])}"] \arrow[r, bend right=50, "{([v_1],\ldots,[v_n])}"'] \arrow[r, phantom, "\Downarrow {([\alpha_1],\ldots,[\alpha_n])}"] & n \arrow[r, bend left=50, "{[t]}"] \arrow[r, bend right=50, "{[t']}"'] \arrow[r, phantom, "\Downarrow {[\beta]}"] & 1
\end{tikzcd}
$$

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 [74], $\mathcal{L}_{\mathcal{R}}$ 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 the category of canonical 2-product-preserving 2-functors from $\mathcal{L}_{\mathcal{R}}$ to $\mathbf{Cat}$.

Therefore, models of rewriting logic have a natural 2-categorical interpretation in which $\mathcal{L}_{\mathcal{R}}$ plays the role of 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 [25] 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 [98]. 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.

More recently, Gadducci and Montanari have proposed another generalization of $\mathcal{L}_{\mathcal{R}}$ to a *double category* [45], in which one composes vertically and horizontally square "tiles" instead of just cells. Their model is particularly perspicuous for dealing with *context conditions* required for a rewriting, such as those arising in structural operational semantics presentations. In particular, they give a very elegant categorical semantics of CCS that fits very precisely with its standard operational semantics definiton.

6.2 Infinite Computations

The concurrent computations described by the proofs of a rewrite theory $\mathcal{R}$ are all *finite*. However, for some purposes such as reasoning about the fairness properties of a system one also wants to consider infinite computations. Somehow, they are already available in the models of $\mathcal{R}$, for example as functors $\omega \longrightarrow \mathcal{T}_{\mathcal{R}}$, where ω is the poset of the natural numbers in ascending order. Such functors intuitively correspond to *chains* of finite "snapshots" of the progress of the computation along time.

This viewpoint can be made more abstract by considering adequate notions of equivalence between such chains, since the particular instant chosen for each snapshot should not matter. Sassone, Montanari and I have studied in detail this more abstract notion of infinite computation in the particular case when $\mathcal{R}$ is the rewrite theory of a Petri net [93]. However, the techniques that we have developed, based on the notion of completion by filtered colimits of a category, should extend naturally to the case of rewrite theories.

Talcott has focused on the case of infinite computations for actor systems specified in rewriting logic, to obtain a precise semantic account of their fair computations. Her ideas and results are very elegant; they are discussed in [99].

Yet another, quite interesting approach has been taken by Corradini and Gadducci [24]. They consider rewrite theories with empty set of equations and interpret them in *continuos* cpo algebra models in such a way that not only the terms, but also the proof terms, become endowed with an approximation ordering. They then propose a natural *infinitary* extension of rewriting logic that allows reasoning about infinite computations that are limits of finite ones. Then, they study the soundness and completeness of their infinitary logic in terms of 2-category models in an adequate cpo-enriched 2-category.

6.3 Formal Reasoning, Refinement, and Program Transformation

Although a good amount of formal reasoning about rewriting logic specifications can be carried out within the logic itself—a good example is the formal reasoning about actor systems illustrated by Talcott in [100]—one may want for certain purposes, such as for example to reason about global properties of a system, to use a more abstract formal language to complement the more operational formalization provided by rewriting logic. Modal or temporal logics seem good candidates for this more abstract level of specification. Lechner and Lengauer [56] and more recently Lechner [55] have proposed the modal μ-calculus as that more abstract level of specification. Modal μ-calculus specifications can then be *refined* into more specific ones until reaching a rewriting logic specification. Martí-Oliet and I are also investigating an adequate modal logic for similar purposes.

In fact, the refinement process is also very important at the level of rewriting logic specifications, since it 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 theories was proposed in [74]. A similar notion has also been proposed and used by Lechner Lengauer and Wirsing in [58]. 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 [32].

Viry has developed a very useful program transformation techniques for rewrite theories using completion methods [102]. 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 [62].

6.4 Rewriting Logic as a Logical Framework

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. Doing justice to this logical side is beyond the scope of this paper, but good evidence, including a good number of examples of logic representations can be found in two joint papers with Martí-Oliet [68, 69]. Additional quite impressive evidence is also provided by research based on the ELAN language [46, 103, 49, 14, 18], that stresses the logical framework applications of rewriting logic.

There is also a very fruitful relationship between rewriting logic and the theory of *reasoning theories* proposed by Giunchiglia, Pecchiari and Talcott [39]. 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 [82].

The work of Levy and Agustí [60, 59, 61] and the more recent work of Schorlemmer [95] explores the relationships between rewriting logic and their general bi-rewriting approach to automated deduction.

6.5 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 [23] 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 in this precise sense. This opens up very interesting possibilities for rewiting logic languages that will be exploited in Maude [22], ELAN [48], and Cafe.

Reflection is closely connected with the topic of *strategies* that is of outmost importance in rewriting logic to control the rewriting process. Clavel and I have proposed the notion of an *internal strategy language* for a general logic, and have advocated it for rewriting logic [23] as a way of being able to reason formally within the logic about the semantics of strategies. This general idea of expressing strategies with rewrite rules has also been adopted by the most recent work on ELAN [15], and by the Maude system [22].

6.6 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, have illustrated those advantages with many examples, and have shown how other logical approaches to dynamic change can be subsumed within rewriting logic [70].

6.7 Nondeterminism

A poset is a poor man's category. Therefore, the different algebraic powerset models of nondeterminism can be understood as categories. In this way, as explained in [75], many models of nondeterminism can be viewed as restricted instances of models of rewrite theories. The relationship between rewriting logic and algebraic models of nondeterminism is further explored in the recent survey on the subject by Meldal and Walicki [104].

7 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.

We can see the outlines of rewriting logic as a wide-spectrum semantic framework. This framework will be complemented at a more abstract level by specifications in some form of temporal or modal logic that can be further refined through formal techniques, first into rewriting logic specifications, and then into efficent declarative programs in subsets for which good parallel implementations exist. Encouraging advances at all the levels of this spectrum have already been made. However, much work remains ahead to tie these levels together, and to develop both formal techniques and well-finished tools supporting all the tasks involved. Tools of this kind include verification tools to mechanically check important properties, sequential and parallel compilers, and tools for program transformation. The area of reflection and strategies seems very important and promising, and will also require much more work.

With 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. Areas such as parallel and distributed programming, mobile computing, office automation, hardware verification, parallel symbolic simulation, logical framework uses, and formal software environments would provide very good application opportunities. All this will keep us busy for a while.

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, Manuel Clavel, Steven Eker, Kokichi Futatsugi, Jean-Pierre Jouannaud, Patrick Lincoln, Narciso Martí-Oliet, 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 Ugo Montanari. Manuel Clavel, Jagan Jagannathan, Patrick Lincoln, Peter Ölveczky, and Carolyn Talcott deserve special thanks for their help reading earlier drafts of this paper. I finally thank again the organizers of CONCUR'96 for giving me the opportunity of presenting these ideas in Pisa, a beautiful city full of dear friends and fond memories.

References

1. Peter 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.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Workshop on Theory of Hybrid Systems*, pages 209–229. Springer LNCS 739, 1993.
6. Rajeev Alur and David Dill. The theory of timed automata. In J.W. 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. J.-P. Banâtre and D. Le Mètayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.
8. E. Battiston, V. Crespi, F. De 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.
9. M. Bauderon and B. Courcelle. Graph expressions and graph rewriting. *Math. Systems Theory*, 20:83–127, 1987.
10. Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
11. E. Best and R. Devillers. Sequential and concurrent behavior in Petri net theory. *Theoretical Computer Science*, 55:87–136, 1989.
12. 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.
13. 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.
14. 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.
15. P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
16. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
17. G. Boudol. Computational semantics of term rewriting systems. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 169–236. Cambridge University Press, 1985.
18. C. Castro. An approach to solving binary CSP using computational systems. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
19. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
20. A. Chien. *Concurrent Aggregates.* MIT Press, 1993.
21. A. Ciampolini, E. Lamma, P. Mello, and C. Stefanelli. Distributed logic objects: a fragment of rewriting logic and its implementation. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
22. Manuel G. Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
23. Manuel G. Clavel and José Meseguer. Axiomatizing reflective logics and languages. In Gregor Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.
24. A. Corradini and F. Gadducci. CPO models for infinite term rewriting. In *Proc. AMAST'95*, pages 368–384. Springer LNCS 936, 1995.
25. 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.
26. Andrea Corradini and Ugo Montanari. An algebra of graphs and graph rewriting. In D.H. Pitt et al., editor, *Category Theory and Computer Science*, pages 236–260. Springer LNCS 530, 1991.
27. 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.
28. P. Degano, J. Meseguer, and U. Montanari. Axiomatizing net computations and processes. In *Proc. LICS'89*, pages 175–185. IEEE, 1989.
29. P. Degano, J. Meseguer, and U. Montanari. Axiomatizing the algebra of net computations and processes. To appear in *Acta Informatica*, 1996.
30. 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.
31. 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.
32. R. Diaconescu. Hidden sorted rewriting logic. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
33. H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Graph Grammars and their Application to Computer Science*. Springer LNCS 532, 1991.
34. Steven Eker. Fast matching in combination of regular equational theories. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
35. J. Engelfriet, G. Leih, and G. Rozenberg. Parallel object-based systems and Petri nets, I and II. Technical Report 90-04,90-05, Dept. of Computer Science, University of Leiden, February 1990.
36. J. Engelfriet, G. Leih, and G. Rozenberg. Net-based description of parallel object-based systems, or POTs and POPs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, Noordwijkerhout, The Netherlands, May/June 1990*, pages 229–273. Springer LNCS 489, 1991.
37. K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. To appear in *Proc. of the Kunming International CASE Symposium, Kunming, China, November*, 1994.
38. F. Gadducci and U. Montanari. Enriched categories as models of computation. In *Proc.* 5^{th} *Italian Conference on Theoretical Computer Science, Ravello*, 1995.
39. F. Giunchiglia, C.L. Pecchiari, and C. Talcott. Reasonig theories: towards an architecture for open mechanized reasoning systems. Technical Report 9409-15, IRST, University of Trento, November 1994.
40. Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992. Originally given as lecture at *Seminar on Types*, Carnegie-Mellon University, June 1983; several draft and technical report versions were circulated since 1985.
41. T.A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J.W. 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.
42. ISO. *IS8807 : Information Processing Systems - Open System Interconnection - LOTOS - A formal description technique based on the temporal ordering of*

observational behavior. ISO, February 1989.

43. R. Jagannathan. Dataflow models. In E.Y. Zoyama, editor, *Parallel and Distributed Computing Handbook*, pages 223–238. McGraw Hill, 1996.
44. R. Jungclaus, 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.
45. G.M. Kelly and R. Street. Review of the elements of 2-categories. In G.M. Kelly, editor, *Category Seminar, Sydney 1972/73*, pages 75–103. Springer Lecture Notes in Mathematics No. 420, 1974.
46. 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.
47. 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.
48. H. Kirchner and P.-E. Moreau. Computational reflection and extension in ELAN. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
49. H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proc. Rewriting Techniques and Applications, Kaiserslautern*, 1995.
50. P. Kosiuczenko and M. Wirsing. Timed rewriting logic, 1995. Working material for the 1995 Marktoberdorf International Summer School "Logic of Computation".
51. C. Landauer. Discrete event systems in rewriting logic. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
52. 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.
53. C. Laneve and U. Montanari. Axiomatizing permutation equivalence. *Mathematical Structures in Computer Science*, 1994. To appear.
54. F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences*, 50:869–873, 1963. Summary of Ph.D. Thesis, Columbia University.
55. U. Lechner. Object-oriented specification of distributed systems in the μ-calculus and Maude. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
56. U. Lechner and C. Lengauer. Modal μ-Maude. To appear in *Object Orientation with Parallelism and Persistence*, B. Freitag, C.B. Jones, C. Lengauer and H.-J. Schek, editors, Kluwer, 1996.
57. U. Lechner, C. Lengauer, F. Nickl, and M. Wirsing. How to overcome the inheritance anomaly. To appear in *Proc.ECOOP'96*, Springer LNCS, 1996.
58. 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.

59. 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.
60. 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.
61. 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.
62. Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.
63. Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Livio Ricciulli. Compiling rewriting onto SIMD and MIMD/SIMD machines. In *Proceedings of PARLE'94, 6th International Conference on Parallel Architectures and Languages Europe*, pages 37–48. Springer LNCS 817, 1994.
64. Patrick Lincoln, José Meseguer, and Livio 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.
65. R.P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, April 1987.
66. The Chemical Abstract Machine. Gérard Berry and Gérard Boudol. In *Proc. POPL'90*, pages 81–94. ACM, 1990.
67. Saunders MacLane. *Categories for the working mathematician*. Springer-Verlag, 1971.
68. Narciso Martí-Oliet and José 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*, Oxford University Press.
69. Narciso Martí-Oliet and José Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.
70. Narciso Martí-Oliet and José Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhoefer, editors, *Theoretical Approaches to Dynamic Worlds in Artificial Intelligence and Computer Science*. 1996. To be published by Kluwer Academic Publisher.
71. Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
72. José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.
73. José Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of the Concur'90 Conference, Amsterdam, August 1990*, pages 384–400. Springer LNCS 458, 1990.

74. José Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, February 1990. Revised June 1990.
75. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
76. José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
77. José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar M. Nierstrasz, editor, *Proc. ECOOP'93*, pages 220–246. Springer LNCS 707, 1993.
78. José Meseguer, Kokichi Futatsugi, and Timothy Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. In *Proceedings of the 1992 International Symposium on New Models for Software Architecture, Tokyo, Japan, November 1992*, pages 61–106. Research Institute of Software Engineering, 1992.
79. José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, 1993.
80. José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.
81. José Meseguer and Xiaolei Qian. A logical semantics for object-oriented databases. In *Proc. International SIGMOD Conference on Management of Data*, pages 89–98. ACM, 1993.
82. José Meseguer and Carolyn Talcott. Reasoning theories and rewriting logic. Manuscript, Stanford University, June 1996.
83. José Meseguer and Timothy Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.
84. Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
85. U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32:545–596, 1995.
86. 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.
87. Peter Csaba Ölveczky, Piotr Kosiuczenko, and Martin Wirsing. An object-oriented algebraic steam-boiler control specification. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *The Steam-Boiler Case Study Book*. Springer-Verlag, 1996. To appear.
88. Peter Csaba Ölveczky and José Meseguer. Specifying real-time systems in rewriting logic. Paper in preparation.
89. Isabel Pita and Narciso Martí-Oliet. A Maude specification of an object oriented database model for telecomunication networks. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
90. 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.
91. 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.
92. G. Saake and A. Sernadas, editors. *Information Systems—Correctness and Reusability*. Technische Universität Braunschweig, Information-Berichte 91-03, 1991.
93. Vladimiro Sassone, José Meseguer, and Ugo Montanari. Inductive completion of monoidal categories and infinite net computations. Submitted for publication.
94. H.-J. Schneider and H. Ehrig, editors. *Graph Transformations in Computer Science*. Springer LNCS 776, 1994.
95. M. Schorlemmer. Bi-rewriting rewriting logic. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
96. Wolfgang Schreiner. Parallel functional programming: an annotated bibliography. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 1993.
97. M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. vanEekelen, editors. *Term Graph Rewriting*. Wiley, 1993.
98. J.G. 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.
99. C. L. Talcott. An actor rewrite theory. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
100. C. L. Talcott. Interaction semantics for components of distributed systems. In *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96*, 1996.
101. P. Viry. Input/output for ELAN. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
102. 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.
103. M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes*. PhD thesis, Université Henry Poincaré — Nancy I, 1994.
104. M. Walicki and S. Meldal. Algebraic approaches to nondeterminism—an overview. To appear in *Computing Surveys*.
105. M. Wirsing. A formal approach to object-oriented software enginering. To appear in *Proc.* 1^{st} *Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1996.
106. 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