DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume **00**, 0000

Specification, Transformation, and Programming of Concurrent Systems in Rewriting Logic

PATRICK LINCOLN, NARCISO MARTÍ-OLIET, AND JOSÉ MESEGUER

May 1994

ABSTRACT. This paper proposes a declarative paradigm in which parallelism is implicit and machine-independent, and the programs so developed are intrinsically parallel. This paradigm is obtained by generalizing the notion of rewriting to make it more widely applicable and capable of expressing not only functional computations but also a wide variety of parallel computations that are highly nonfunctional in nature. The generalization in question is provided by rewriting logic, a logic of change in which the states of a system are understood as algebraically axiomatized data structures, and the basic local changes that can concurrently occur in a system are axiomatized as rewrite rules that correspond to local patterns that, when present in the state of a system, can change into other patterns. Simple Maude, a carefully designed sublanguage of rewriting logic supporting three types of rewriting—term, graph, and object-oriented—, is then proposed as a machine-independent parallel programming language that can be efficiently implemented in parallel on many different machines. The adequacy of term, graph, and object-oriented rewriting to naturally express many different parallel programming problems is illustrated with examples. Several program transformation techniques mapping rewriting logic specifications into Simple Maude programs are discussed using representative examples. The incorporation of modules containing conventional code or special subsystems or devices within Simple Maude is also discussed. Finally, the advances made so far on general compilation techniques for Simple Maude are summarized.

1991 *Mathematics Subject Classification.* Primary 68Q10, 68N15; Secondary 68Q42, 68Q60.

Supported by Office of Naval Research Contracts N00014-90-C-0086, N00014-92-C-0222, and N00014-92-C-0518, 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

The difficulties and the limited returns often encountered when programming parallel machines with the languages currently available are among the biggest obstacles blocking the widespread acceptance of parallel computing. In many cases users find that parallelizing compilers only achieve modest speedups for running their code on parallel machines (see for example [**37**] for an informative report on experience of this kind). Indeed, given the much higher cost at present of parallel machines compared to advanced workstations, it is only through impressive speedups in performance that most users will find parallel machines attractive.

There is, of course, the alternative of programming an application using a parallel language. However, for some machines this means programming using low-level machine-specific instructions. Even if the programming language used is quite portable across a given class of machines—as is the case for some of the extensions of the C language with message-passing capabilities [**35, 12**] within the class of distributed memory message-passing multicomputers—such a relatively portable language may not be well suited for other architectures such as SIMD machines.

It seems fair to say that programming parallel machines is at present considerably harder than programming sequential machines, and that the resulting programs have at best limited portability. For these reasons, a considerable gap often exists between users with applications needing fast, efficient, solutions and the machines that have, at least in principle, the raw power to solve them. In some cases bridging the gap requires employing programmers who have expertise programming a particular parallel machine, but who need to acquire detailed knowledge of the user's application area in order to correctly program a good solution.

The complexity and relative lack of portability currently exhibited by most parallel programming languages are both due to the *explicit* way in which parallelism is programmed. Such explicit parallelism further complicates an already complex sequential language with additional constructs, and unavoidably builds in some architectural assumptions about the machines on which the language is supposed to run.

1.1. A machine-independent declarative paradigm. The alternative we have in mind to bridge the existing software gap in parallel computing is to allow users to express their problem in a declarative way that is as close as possible to the concepts of the application area in question. In this paradigm, parallelism is *implicit* and machine-independent, and the programs so developed are intrinsically parallel from the start.

In this regard, rewriting, that is, the process of replacing instances of a left-hand side pattern by corresponding instances of a righthand side pattern, has been recognized as an intrinsically concurrent computational paradigm. The application of a rewrite rule only depends on the local existence of a pattern so that rules can be applied simultaneously in many places. In particular, rewrite rules have been used for expressing the implicit parallelism of functional programs in a declarative way. This has led to the investigation of so-called *reduction* archi-

tectures that try to exploit this type of parallelism (see for example [**16, 33**]).

There are, however, many applications that are certainly amenable to parallelization but do not fit well within the functional paradigm. For example, a discrete-event simulation in which many different objects interact with each other may have a high degree of concurrency, which can be exploited using optimistic parallel simulation methods such as the Time Warp [**24**]. However, this usually does not have a natural formulation as a functional program. Efficiency considerations may also discourage the functional formulation of problems for which nonfunctional in-place data replacements can be considerably more efficient. In general, there are many problems and algorithms that are state-oriented or nondeterministic in such an intrinsic way that thinking of them as the computation of a functional expression is both implausible and ill-advised.

In this paper we explain how, by adequately generalizing the notion of rewriting, we can arrive at a declarative and machine-independent parallel programming paradigm that is indeed widely applicable, and can therefore express not only functional computations but also a very wide variety of other parallel computations that are highly nonfunctional in nature. The generalization in question is provided by *rewriting logic* [**28**], a logic of change in which the states of a system are understood as algebraically axiomatized data structures, and the basic local changes that can concurrently occur in a system are axiomatized as rewrite rules that correspond to local patterns that, when present in the state of a system, can change into other patterns.

Formally, viewing a state as an algebraically axiomatized data structure means that we give a set of equational axioms E that capture the *structural* properties of states of this kind. Then, a given state of the system we are interested in can be represented as an equivalence class $[t]$ of a certain term t *modulo* the structural axioms E. The rewrite rules axiomatizing concurrent computation are then expressions of the form

$$[t] \longrightarrow [t']$$

where $[t]$ and $[t']$ are equivalence classes modulo E of terms t and t' (that can contain variables) describing respectively the local pattern to be found and its replacement. Concurrent computation in a system so described can be formalized as concurrent rewriting modulo the structural axioms E; in rewriting logic each such computation exactly corresponds to a *proof* in the logic.

The case of parallel functional programming can then be recovered as that in which the rewrite rules are *confluent* (also called Church-Rosser) so that no matter how they are applied, the final result, if it exists, is always unique. However, in many other applications the rules need not be confluent and need not terminate; this reflects the intrinsic nondeterminism, and in some cases the "reactive" character, of the system in question.

1.2. Maude and Simple Maude. 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

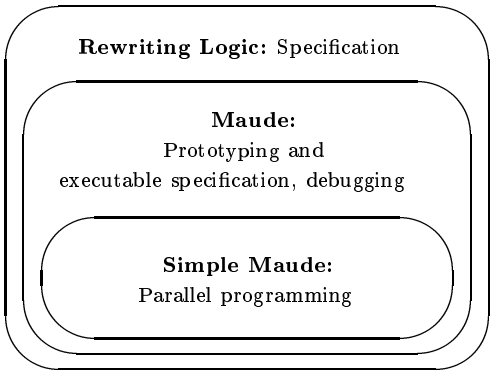

FIGURE 1. Maude and Simple Maude as subsets of rewriting logic.

pattern modulo E, gives rise to the Maude language [**31, 29**], in the sense that Maude modules are rewriting logic theories in that subset, and can be supported by an interpreter implementation adequate for rapid prototyping, debugging, and executable specification. The second, smaller subset gives rise to Simple Maude, a sublanguage meant to be used 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 1 summarizes the three levels involved.

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.

For cases in which substantial efforts have already been spent developing conventional sequential code for some parts of an application, or interaction with special-purpose devices needs to be included in a parallel program, Simple Maude provides a simple way of developing parallel applications. Simple Maude programs can incorporate conventional or special-purpose subcomponents by encapsulating them as *foreign interface modules* that act as black boxes with which other modules can interact in a concurrent message-passing 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.

1.3. Transforming specifications and programs. An important advantage of having Maude and Simple Maude as increasingly more restrictive subsets of rewriting logic is that formal techniques can be applied within the logical framework of rewriting logic to derive Maude prototypes from rewriting logic specifications, Simple Maude programs from Maude prototypes, and more efficient Simple Maude programs from less efficient ones in a manifestly correct fashion.

In Section 5 we illustrate three program transformation techniques with examples. One technique, due to Viry [**38**], transforms a rewrite theory with rules R and structural axioms E into an equivalent theory with rules R' and structural axioms E' such that rewriting modulo E' is directly implementable in Maude. A second technique applies to Maude object-oriented modules whose rewrite rules may involve several objects in their lefthand side, that is, the rules require *object synchronization*; such object-oriented modules can be transformed into equivalent Simple Maude modules whose rules only involve asynchronous message passing. A third technique applies to graph rewrite rules that require synchronous application everywhere—typically achieved by a globally SIMD implementation—for their correctness; such rules can be transformed into more flexible rules that do not require simultaneous global application and that can be implemented in an asynchronous MIMD/SIMD regime.

1.4. Outline of this paper. The paper is organized as follows. Section 2 introduces the more familiar term rewriting case and illustrates its intrinsic parallelism using a symbolic computation example. Section 3 motivates the inadequacy of equational logic as a general framework for rewriting and introduces rewriting logic by means of object-oriented rewriting examples as the resolution

of this inadequacy. Section 4 illustrates graph rewriting using an image labelling example. Section 5 presents the three program transformation techniques discussed above. Section 6 discusses how conventional programs, subsystems, and special hardware devices can be integrated in Simple Maude by means of *foreign interface modules*. Section 7 gives a taxonomy of parallel machine architectures and summarizes the general compilation techniques that have been developed so far. The paper ends with some concluding remarks.

2. Term rewriting

The experience of "replacing equals for equals" in elementary algebra amounts to an early introduction to term rewriting. The intrinsic parallelism of this process follows from the localized nature of each replacement, being independent of any other replacement if they affect different parts of the overall expression. We illustrate this parallelism by means of a simple example of derivatives of polynomials. In this example, the interpretation of rewrite rules is as equations so that indeed we are replacing "equals for equals" in a *functional* module. We shall see later other examples where rewrite rules are *not* equalities.

Assume a set `Var` of variable names. A monomial is a product

```
X1 ^ N1 ... Xk ^ Nk
```

of powers of variables, i.e., the `Xi` are elements of `Var`, and the `Ni` are positive integers. There is a product operation on monomials satisfying, among others, the equation

```
(X ^ N) . (X ^ M) = X ^ (N + M).
```

A polynomial is of the form $(\sum_i A_i * U_i) + C$, where A_i and C are coefficients in a ring of numbers, for example integers, and U_i are monomials as described above. The main operations on polynomials are product and sum. They satisfy, among others, the following equations:

```
P + 0 = P
(A * U) + (B * U) = (A + B) * U
(A * U) . (B * V) = (A B) * (U . V)
```

for a polynomial `P`, monomials `U` and `V`, and coefficients `A` and `B`.

Polynomial derivation takes as arguments a variable (so that one can differentiate with respect to different variables) and a polynomial, producing a polynomial. It can be fully defined by the seven equations given in the functional module[1] `POLY-DER` below, which imports a `POLYNOMIAL` submodule, in the sense that using those equations as rewrite rules from left to right plus rules of simplification for polynomials, the result of differentiating a polynomial with respect to a variable is obtained.

```
fmod POLY-DER is
  protecting POLYNOMIAL .
  op der : Var Poly -> Poly .
  op der : Var Mon -> Poly .
```

[1]The syntax of functional modules is very similar to that of OBJ [**18**, **21**] and for the most part is self-explanatory. They are introduced with the keyword `fmod`, the type of each operator and of each variable is declared, and then the equations are introduced with the keyword `eq` (or `ceq` for conditional equations).

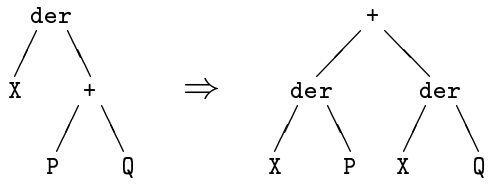

FIGURE 2. A tree rewrite rule.

```
  var A : Int .
  var N : NzNat .
  vars P Q : Poly .
  vars X Y : Var .
  vars U V : Mon .
  eq  der(X, P + Q) = der(X, P) + der(X, Q) .
  eq  der(X, U . V) = (der(X, U) . V) + (U . der(X, V)) .
  eq  der(X, A * U) = A * der(X, U) .
  ceq  der(X, X ^ N) =  N * (X ^ (N - 1)) if N > 1 .
  eq  der(X, X ^ 1) = 1 .
  ceq  der(X, Y ^ N) = 0 if X =/= Y .
  eq  der(X, A) = 0 .
endfm
```

In addition, the seven equations above constitute a *parallel program* for polynomial differentiation because equations can be applied concurrently with other equations whenever they happen to match a subexpression, and the final result is always the same independently of the order in which the equations are applied[2]. This can be visualized by representing all expressions in tree form. For example, the equation

```
  der(X, P + Q) = der(X, P) + der(X, Q)
```

can be represented as the tree rewrite rule in Figure 2.

The concurrent rewriting computation of a polynomial differentiation example can be expressed in graph form as in Figure 3, where the last step summarizes several concurrent steps of polynomial simplification using rules in `POLYNOMIAL` (not shown).

3. Rewriting logic and object-oriented rewriting

The type of rewriting typical of functional programming applications just illustrated in Section 2 can be generalized in two ways. We can:

- allow rewrite rules that can be nonconfluent and/or nonterminating,
- rewrite modulo certain structural axioms satisfied by the data.

Nonconfluence can quickly lead us outside the realm of equational logic, as illustrated by the following nonfunctional term rewriting example which adds a nondeterministic choice operator to the natural numbers.

```
mod NAT-CHOICE is
  extending NAT .
  op _?_ : Nat Nat -> Nat .
```

[2]This property follows from the above equations being *confluent* and *terminating*.

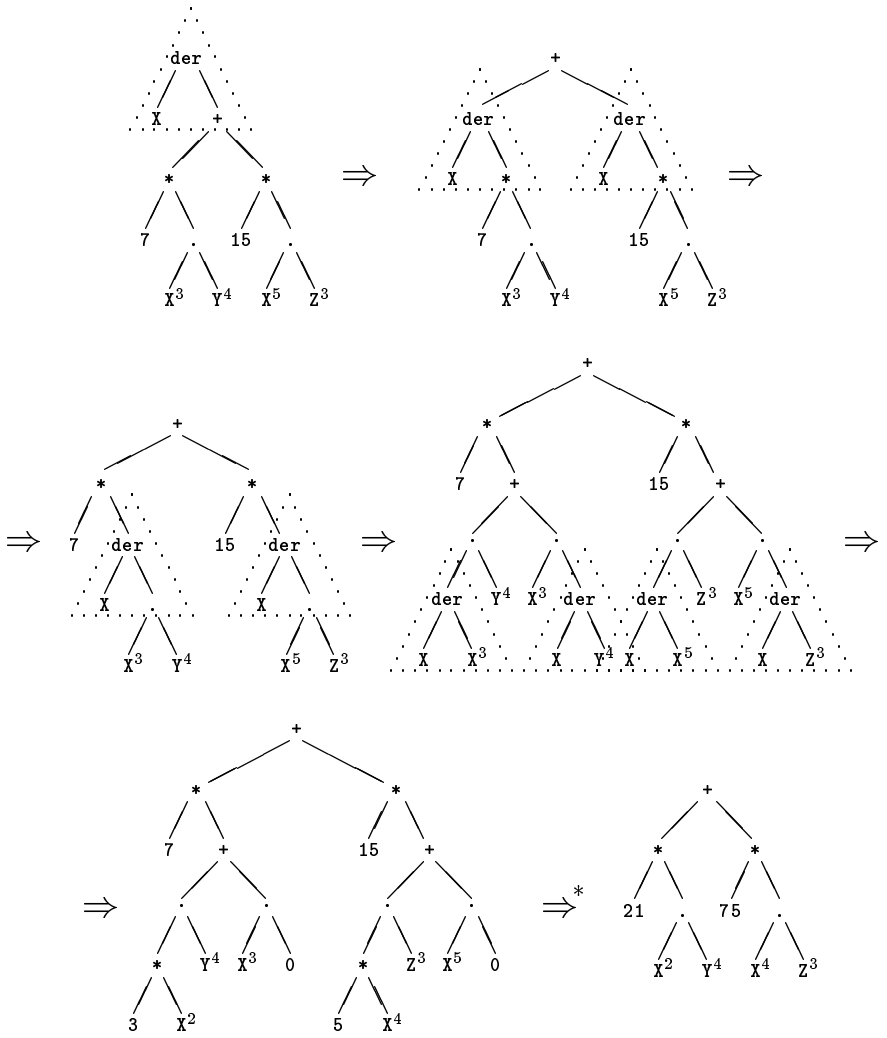

FIGURE 3. Concurrent polynomial differentiation in graph form.

```
  vars N M : Nat .
  rl N ? M => N .
  rl N ? M => M .
endm
```

The intuitive *operational behavior* of this module is quite clear. Natural number arithmetic remains unchanged and is computed using the rules in the `NAT` module (not shown). Using the two rules in the module, any occurrence of the choice operator `_?_` in an expression can be eliminated by choosing either of the arguments. In the end, we can reduce any ground expression to a natural number. However, the rules cannot be interpreted as equalities; otherwise we would obtain the contradiction

```
N  =  N ? M  =  M
```

collapsing all natural numbers into one point. To mark this distinction, the keyword `rl` (for "rule") is used in Simple Maude for rules in nonfunctional modules.

Nonfunctional modules are theories in rewriting logic. We informally introduce rewriting logic by means of a simple object-oriented example. This example, in addition to being nonconfluent, illustrates the importance of rewriting *modulo* a set E of structural axioms. A precise definition of the rules of rewriting logic is given in Appendix A.

Rewriting logic is a logic to reason correctly about the evolution in time of a concurrent system. The distributed state of a concurrent system is represented as a *term* whose subterms represent the different components of the concurrent state. Typically, however, the *structure* of the concurrent state may have a variety of equivalent term representations because it satisfies certain *structural laws*. For example, 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.

As a consequence, we can abstractly represent the configuration of a typical concurrent object-oriented system as an equivalence class $[t]$ modulo the structural laws of associativity, commutativity, and identity obeyed by the multiset union operator of a term expressing a union of objects and messages, i.e., as a multiset of objects and messages.

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 >
```

In rewriting logic, sentences are rewrite rules of the form

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

where $[t]$ denotes the equivalence class of t modulo the structural laws satisfied by the states of the system in question, or, more generally, conditional rewrite rules of the form

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

Those sentences axiomatize the basic *local transitions* that are possible in a concurrent system. For example, in a concurrent object-oriented system including bounded buffers that interact with other objects by `put` and `get` messages, and with appropriate reply messages after a `get`, the local transitions of bounded buffers are axiomatized by rewrite rules in the module below[3].

```
omod BD-BUFF is
  protecting NAT .
  protecting LIST[Nat] .
  class BdBuff | contents: List, in: Nat, out: Nat .
  initially contents: nil, in: 0, out: 0 .
  msg put_in_ : Nat OId -> Msg .
  msg getfrom_replyto_ : OId OId -> Msg .
  msg to_elt-in_is_ : OId OId Nat -> Msg .
  vars B I : OId .
  vars E N M : Nat .
  var Q : List .
  rl (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 .
```

[3]Object-oriented modules in Simple Maude have special syntax facilitating their definition. The existence of a configuration multiset of objects and messages is already assumed and therefore is left implicit. The multiset union operator is indeed used in the two rewrite rules given. Note that the attributes of a class and their types are declared after the class itself. Although Maude provides convenient syntax for object-oriented modules, they can be systematically translated into rewrite theories by making explicit all the assumptions left implicit in their syntax. A detailed account of this translation process can be found in [**29**].

```
  rl (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) .
endom
```

We assume an already defined functional module `NAT` for natural numbers, with arithmetic operations and ordering predicates.

The first rule specifies the conditions under which a `put` message can be accepted (namely, that `N - M` is smaller than `bound`) and the corresponding effect. The second rule does the same for `get` messages; note that the requirement that the buffer must not be empty is implicit in the pattern `Q E` for the `contents` attribute.

The rules of deduction of rewriting logic support sound and complete reasoning about the concurrent transitions that are possible in a concurrent system whose basic local transitions are axiomatized by given rewrite rules. That is, the sentence $[t] \longrightarrow [t']$ is provable in the logic using the rewrite rules that axiomatize the system as axioms if and only if the concurrent transition $[t] \longrightarrow [t']$ is possible in the system. A precise account of the model theory of rewriting logic fully consistent with the above system-oriented interpretation, and proving soundness, completeness, and the existence of initial models is given in [**28**].

The intuitive idea behind the rules of rewriting logic in Appendix A is that proofs in rewriting logic exactly correspond to concurrent computations in the concurrent system being axiomatized, and that such concurrent computation can be understood as concurrent rewriting *modulo* the structural laws obeyed by the concurrent system in question. In the case of a concurrent object-oriented system such structural laws include the associativity, commutativity, and identity of the union operators `__` and `_,_`, and this means that the rules can be applied regardless of order or parentheses. For example, 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[4] 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 supported by Simple Maude. In general, for concurrent object-oriented modules in Simple Maude we only allow conditional

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

rules of the form

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

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

Furthermore, the lefthand sides in rules of the form (†) should fit the general pattern

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

where O could be a variable, a constant, or more generally—in case object identifiers are endowed with additional structure—a term. Under such circumstances, an efficient way of realizing rewriting modulo associativity and commutativity by communication is available to us for rules of the form (†), 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 address.

More general rewrite rules corresponding to synchronous communication between objects that do not satisfy the (†) restriction can be transformed into simpler asynchronous rules of the form (†) by program transformation techniques, as discussed in Section 5.

Support for multiple inheritance for classes is provided by the order-sorted type structure of rewriting logic [**19**]—so that if C is a subclass of C', then C is a *subsort* of C'—and by an associated desugaring of the rules as originally given by the user that makes them automatically applicable in subclasses. See [**29, 30**] for more details on the semantics of multiple inheritance for object-oriented modules in Maude and Simple Maude.

Our above introduction to rewriting logic has focused on the object-oriented case where the structural axioms E are the associativity, commutativity, and identity of a multiset union operator that builds up the configuration of objects and messages. In general, however, the axioms E can be varied as a very flexible parameter with which many different types of concurrent systems can be naturally specified. In this way, rewriting logic can be regarded as a very general model of concurrency from which many other models can be directly obtained by specialization. Examples of such special cases include: labelled transition systems; parallel functional programming, including equational programming and

the λ-calculus with explicit substitution; Post systems and related grammar formalisms; concurrent object-oriented programming, including the Actor model [**4**]; Petri nets [**34**]; the Gamma language of Banâtre and Le Mètayer [**5**], and Berry and Boudol's chemical abstract machine [**7**]; CCS [**32**]; and Unity's model of computation [**9**]. A detailed discussion of how all these models appear as special cases can be found in [**28**], and for CCS in [**27**].

4. Graph rewriting

Graph rewriting is an area that has received much attention and has been used for a variety of purposes (see for example [**15, 36**] and references there). A number of different axiomatizations of graph rewriting for somewhat different variants of the general concept have been proposed in the literature, including axiomatizations in terms of categorical pushouts [**15**]. For our purposes the axiomatizations that provide a direct link with the rewriting logic approach 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 [**6**], and of Corradini and Montanari [**11**].

Among other applications, graph rewriting has been extensively used in the compilation of functional languages. However, the importance of graph rewriting goes far beyond functional applications. For example, many highly regular computations can be naturally expressed by graph rewrite rules in which the topology of the graph does not change. This fixed-topology subcase, besides being quite common in applications such as scientific computing, systolic algorithms, signal processing, and cellular automata, has many advantages allowing very efficient implementations, including the predictability at compile time of communication requirements—which can be minimized by appropiate placement of the data graph—and the lack of any need for garbage collection or for structure creation.

Consider for example the problem of clustering 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 4 shows one such image and its two connected components.

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 5, 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.

This one rewrite rule embodies an algorithm for computing connected components with worst-case sequential complexity of $\mathcal{O}(n^4)$, where the input image is a square with side n and n^2 total pixels. This bound is achieved on the

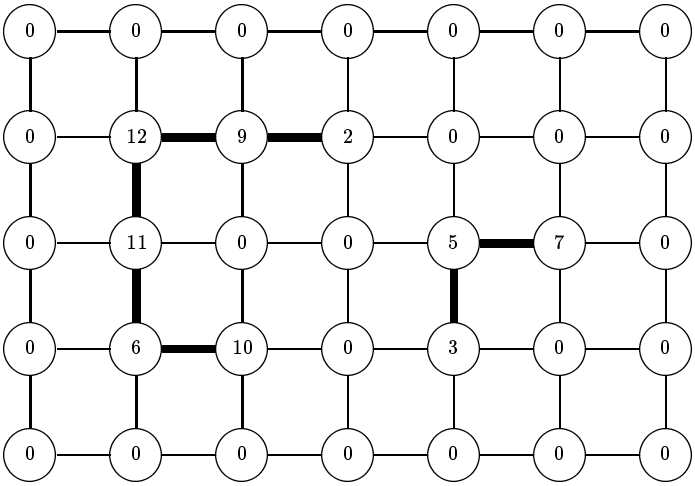

FIGURE 4. Image as two-dimensional array.

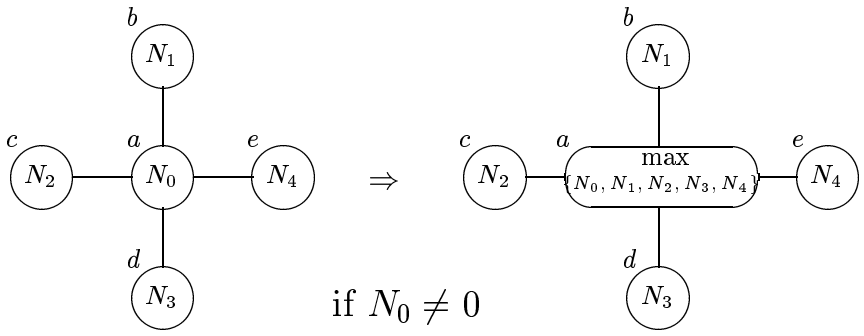

FIGURE 5. A graph rewrite rule.

surprisingly bad case of an image of a spiral. The parallel complexity of this algorithm is $\mathcal{O}(n^2)$, but there exist algorithms with parallel complexity $\mathcal{O}(n \log n)$ that avoid slavishly following paths around complicated diagrams [**13**]. The algorithm above with disappointing worst-case behavior has smaller constant factors, and good average case performance. In any case, an algorithm with optimal worst-case performance can be expressed in a similar way as a small set of graph rewrite rules.

5. Transforming specifications and programs

Three program transformation techniques, namely: coherence completion, reduction of synchronous object communication to asynchronous message passing, and transformation of synchronous graph rewriting into an asynchronous version, are illustrated with examples.

5.1. From rewriting logic to Maude through coherence completion. As discussed in the introduction, rewriting modulo an arbitrary set of structural axioms E may be inefficient and even undecidable in general. However, there are cases in which E satisfies special properties in such a way that rewriting modulo E can be implemented by standard rewriting, or by rewriting modulo an equational theory A for which there are suitable matching and unification algorithms, like for example associativity and commutativity. This subject has been studied by P. Viry in [**38**], whose main results are summarized here.

Viry's results can be used to semiautomate a transformation technique in which a rewriting logic specification given by a set of rewrite rules R modulo a set of equations E can be transformed into either:

(i) An equivalent Simple Maude term rewriting program with rules $R' \cup E'$ (and no structural axioms) such that the rules E' are confluent and terminating and are equivalent to the equations E, and such that rewriting modulo E with the original rules R can be simulated by standard term rewriting using the rules $R' \cup E'$ in the transformed program, or

(ii) An equivalent Maude program with rules $R' \cup E'$ and structural axioms A (for which matching algorithms modulo A exist in the Maude implementation) such that the equations E are equivalent to $E' \cup A$, and such that rewriting modulo E with the original rules R can be simulated by rewriting modulo A using the rules $R' \cup E'$ in the transformed program.

We need to introduce some notation. Let us denote by $\longrightarrow\!\!\!\!\!\rightarrow$ the transitive and reflexive closure of a relation $\longrightarrow$, and by $\longleftrightarrow\!\!\!\!\!\rightarrow$ the equivalence relation generated by $\longrightarrow$ (its symmetric, transitive, and reflexive closure). The composition of relations $\longrightarrow$ and $\rightsquigarrow$ is written $\longrightarrow \bullet \rightsquigarrow$. Then, a step of rewriting modulo E using the rules in R on both equivalence classes ($[R]_E$) and terms (R/E) can be defined as

$$[t]_E \xrightarrow{[R]} [t']_E \quad \Longleftrightarrow \quad t \xrightarrow{R/E} t' \quad \Longleftrightarrow \quad t \overset{E}{\longleftrightarrow\!\!\!\!\!\rightarrow} \bullet \xrightarrow{R} \bullet \overset{E}{\longleftrightarrow\!\!\!\!\!\rightarrow} t'$$

There is also a weaker relation $\xrightarrow{R,E}$ which is the restriction of $\overset{E}{\longleftrightarrow\!\!\!\!\!\rightarrow} \bullet \xrightarrow{R}$ obtained by allowing the $\overset{E}{\longleftrightarrow\!\!\!\!\!\rightarrow}$ steps to be applied only below the redex rewritten by R. This relation can be implemented using an algorithm for matching modulo E.

Assume that the equations E are (or can be completed to be) confluent and terminating as rewrite rules. Then, we want to implement rewriting modulo E using R by a combination of (standard) rewriting using E and rewriting using R. Since we are going to apply R to terms instead of equivalence classes, we must check that the choice of a representative in a class has no effect in the result. This is the case when R and E satisfy the following condition.

DEFINITION 5.1. *We say that* $\xrightarrow{R}$ *is* coherent *with* $\xrightarrow{E}$ *if, when* $t \xrightarrow{R} t'$ *and* $t \overset{E}{\longrightarrow\!\!\!\!\!\rightarrow} t''$, *then there exists* s *such that* $t' \overset{E}{\longrightarrow\!\!\!\!\!\rightarrow} s$ *and* $t'' \overset{E}{\longrightarrow\!\!\!\!\!\rightarrow} \bullet \xrightarrow{R} \bullet \overset{E}{\longrightarrow\!\!\!\!\!\rightarrow} s$.

The coherence condition can be represented diagrammatically as in Figure 6, where solid arrows denote the given rewrite steps, and dotted arrows denote the rewrite steps that have to exist. A similar condition using normal forms was independently proposed by Meseguer in [**29**, p. 359].

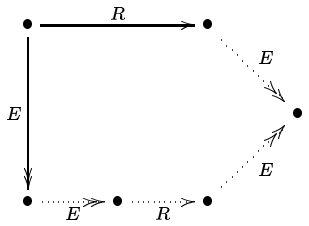

FIGURE 6. Coherence condition in diagram form.

Viry's main result is the following

THEOREM 5.1. *Assume that* $\xrightarrow{E}$ *is confluent and terminating. If* $\xrightarrow{R}$ *is coherent with* $\xrightarrow{E}$*, then the following equivalence holds:*

$$[t]_E \xrightarrow{[R]} [t']_E \quad \Longleftrightarrow \quad t \xrightarrow{E}\!\!\!\!\!\rightarrow \bullet \xrightarrow{R} t'' \in [t']_E.$$

Moreover he proves that coherence can be semiautomatically generated and can be checked by means of a Knuth-Bendix style completion method using critical pairs of coherence; see [**38**] for details. This completion method provides the transformation (i) from rewriting logic specifications into Simple Maude programs.

These results can be generalized to the case in which, although E is not confluent and terminating, it is equivalent to E', which is confluent and terminating *modulo* a set of equations $A \subset E$. For example, A may consist of associativity and commutativity axioms for some operators, or more generally of any axioms for which suitable matching algorithms exist. The presentation of this case requires the definition of new coherence concepts for which the reader is referred to Viry's paper [**38**]. We just repeat here his main result in this more general setting.

THEOREM 5.2. *Assume that* $\xrightarrow{E'/A}$ *is terminating, that* $\xrightarrow{E',A}$ *is Church-Rosser, and that the set of equations* A *is linear, regular, and non-collapsing. If* $\xrightarrow{R,A}$ *is coherent with* $\xleftrightarrow{A}$ *and with* $\xrightarrow{E',A}$*, then the following equivalence holds:*

$$[t]_E \xrightarrow{[R]} [t']_E \quad \Longleftrightarrow \quad t \xrightarrow{E',A}\!\!\!\!\!\rightarrow \bullet \xrightarrow{R,A} t'' \in [t']_E.$$

For axioms A having reasonable matching and unification algorithms, the previous coherence completion method can be extended to a coherence completion method modulo A [**38**]. Provided that a Maude implementation supports matching modulo A, this completion technique can then be employed to semiautomate the transformation (ii) from a rewriting logic specification into a Maude executable specification. We illustrate this second case of transformation by means of the following example representing in rewriting logic a subset of Milner's CCS [**32**].

```
fth NAMES is
  sort ProcessId .     *** process identifiers
  sort Label .         *** ordinary actions
```

```
  op ~_ : Label -> Label .
  var N : Label .
  eq ~~N = N .
endft
fmod PROCESS[X :: NAMES] is
  sort Act .
  subsort Label < Act .
  op tau : -> Act .       *** silent action
  sort Process .
  subsort ProcessId < Process .
  op 0 : -> Process .                      *** inaction
  op _._ : Act Process -> Process .        *** prefix
  op _+_ : Process Process -> Process      *** summation
                      [assoc comm idem id: 0] .
  op _|_ : Process Process -> Process      *** composition
                      [assoc comm id: 0] .
endfm
mod SUB-CCS[X :: NAMES] is
  protecting PROCESS[X] .
  sort ActProcess .
  subsort Process < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess .
    *** {A}P means that process P has performed action A
  var L : Label .
  var A : Act .
  vars P P' Q Q' : Process .
  rl A . P => {A}P .
  crl P + Q => {A}P'  if  P => {A}P' .
  crl P | Q => {A}(P' | Q)  if  P => {A}P' .
  crl P | Q => {tau}(P' | Q')  if  P => {L}P'  and  Q => {~L}Q' .
endm
```

Therefore, making explicit the axioms declared with specific operators, the set E of structural axioms that processes must satisfy is:

```
  eq ~~N = N .
  eq P + Q = Q + P .
  eq (P + Q) + R = P + (Q + R) .
  eq P + 0 = P .
  eq P + P = P .
  eq P | Q = Q | P .
  eq (P | Q) | R = P | (Q | R) .
  eq P | 0 = P .
```

and the set of rules R consists of the four rules in module `SUB-CCS[X]`. Except for the two associativity and commutativity equations, the rest can be oriented from left to right giving a set of rewrite rules which is terminating and confluent modulo associativity and commutativity. That is, we can split E into the set A of structural axioms

```
  eq P + Q = Q + P .
```

```
eq (P + Q) + R = P + (Q + R) .
eq P | Q = Q | P .
eq (P | Q) | R = P | (Q | R) .
```

and the set E' of rules

```
rl ~~N => N .
rl P + 0 => P .
rl P + P => P .
rl P | 0 => P .
```

Moreover, these sets satisfy all the requirements in Theorem 5.2 [**38**], and therefore, rewriting by the rules in R modulo E is equivalent to rewriting by the rules in $R \cup E'$ modulo A.

5.2. Transforming synchronous object communication into asynchronous message passing. In Section 3 we have described rules for Simple Maude object-oriented modules of the form (†), involving only at most one object and one message in the lefthand side. More generally, as mentioned there, we can consider synchronous rules which involve several objects and messages in the lefthand side of the form

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

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. Although synchronous rules of this kind could be implemented in parallel, their direct implementation would be very communication intensive and therefore would be inefficient. For this reason, they are not allowed in Simple Maude (where only asynchronous message passing communication between objects is directly supported), but they are permitted in Maude object-oriented modules, where they can be implemented in a sequential interpreter using an associative-commutative matching algorithm. In what follows we illustrate with a simple example the transformation of Maude object-oriented modules with synchronous rules (‡) into corresponding Simple Maude modules with asynchronous message passing rules (†).

Consider the object-oriented `SPREADSHEET` module below which specifies the concurrent behavior of objects in a very simple class `Cell` of cells in a spreadsheet, whose unique attribute is the value stored in the cell, set initially to zero. The cells are organized in a grid and are therefore identified by means of pairs `(N,M)` giving the row and column numbers. For each row `N` there is a cell `(N,total)` that keeps track of the corresponding total, and similarly for each column `M` there is a cell `(total,M)`. There is also a cell `(total,total)` providing the sum of all the values in all the cells in the spreadsheet. The spreadsheet may receive messages `add(N,M,V)` and `sub(N,M,V)` for adding or subtracting the amount `V` to the value stored in cell `(N,M)`. We also assume a functional module `NAT` for natural numbers.

The reader can compare this Maude program with the more complex program developed by Chandy and Taylor in [**10**], which stimulated our alternative solution.

```
omod SPREADSHEET is
  protecting NAT .
  sort Name .
  subsort Nat < Name .
  op total : -> Name .
  op (_,_) : Name Name -> OId .
  class Cell | val : Nat .
  initially val : 0 .
  msgs add sub : Nat Nat Nat -> Msg .
  vars M N V W X Y Z : Nat .
  rl add(N,M,V) < (N,M) : Cell | val: W >
                < (total,total) : Cell | val: X >
                < (N,total) : Cell | val: Y >
                < (total,M) : Cell | val: Z >
     => < (N,M) : Cell | val: W + V >
        < (total,total) : Cell | val: X + V >
        < (N,total) : Cell | val: Y + V >
        < (total,M) : Cell | val: Z + V > .

  crl sub(N,M,V) < (N,M) : Cell | val: W >
                 < (total,total) : Cell | val: X >
                 < (N,total) : Cell | val: Y >
                 < (total,M) : Cell | val: Z >
     => < (N,M) : Cell | val: W - V >
        < (total,total) : Cell | val: X - V >
        < (N,total) : Cell | val: Y - V >
        < (total,M) : Cell | val: Z - V >
     if W >= V .
endom
```

The problem we address in this section is how to transform synchronous object-oriented rules of the form (‡), like the ones in the module above, into asynchronous rules of the simpler form (†). The essential idea is to introduce new messages in the righthand side of the rules, creating new states in which the original computation is half-done, and is going to continue by further interaction of the new messages with the objects. In the particular case of the spreadsheet example, we have the following program in Simple Maude.

```
omod SPREADSHEET-ASYNCH is
  protecting NAT .
  sort Name .
  subsort Nat < Name .
  op total : -> Name .
  op (_,_) : Name Name -> OId .
  class Cell | val: Nat .
  initially val: 0 .
```

```
  msgs add sub : Nat Nat Nat -> Msg .
  msgs add-row add-col : Nat Nat -> Msg .
  msgs sub-row sub-col : Nat Nat -> Msg .
  msgs add-total sub-total : Nat -> Msg .
  vars M N V W : Nat .
  rl add(N,M,V) < (N,M) : Cell | val: W >
     => < (N,M) : Cell | val: W + V >
        add-row(N,V) add-col(M,V) add-total(V) .
  rl add-row(N,V) < (N,total) : Cell | val: W >
     => < (N,total) : Cell | val: W + V > .
  rl add-col(M,V) < (total,M) : Cell | val: W >
     => < (total,M) : Cell | val: W + V > .
  rl add-total(V) < (total,total) : Cell | val: W >
     => < (total,total) : Cell | val: W + V > .

  crl sub(N,M,V) < (N,M) : Cell | val: W >
     => < (N,M) : Cell | val: W - V >
        sub-row(N,V) sub-col(M,V) sub-total(V)
     if W >= V .
  rl sub-row(N,V) < (N,total) : Cell | val: W >
     => < (N,total) : Cell | val: W - V > .
  rl sub-col(M,V) < (total,M) : Cell | val: W >
     => < (total,M) : Cell | val: W - V > .
  rl sub-total(V) < (total,total) : Cell | val: W >
     => < (total,total) : Cell | val: W - V > .
endom
```

Because of the presence of new messages, there are new configurations in the module `SPREADSHEET-ASYNCH` that do not correspond to any configuration in the original module `SPREADSHEET`. However, in any computation using the new rules that starts in a configuration from `SPREADSHEET` the new messages are going to eventually disappear by application of the new rules involving those messages on the lefthand side, reaching in this way a configuration in `SPREADSHEET`. Moreover, this configuration is exactly the same achieved by the original synchronous rules.

The spreadsheet example illustrates the main idea of this transformation technique, but it is simpler than usual because the operations involved satisfy special properties, like for example commutativity of addition, that make the order of application of rules irrelevant with respect to the final result. In general, the order in which rules are applied matters, and this has to be taken care of when transforming the program. Transformation techniques covering this general case that can automate the compilation of Maude object-oriented modules into Simple Maude have been developed by Lincoln and Meseguer in joint work with T. Winkler. The main idea of the general-purpose transformation techniques from Maude rules of type (‡) into Simple Maude rules of type (†) is to use one of the objects involved in the rewrite as the locus of control for each rule. The main problematic points are deadlock and fairness. Here we give the basic outline of the approach for Maude rules whose lefthand side contains only one message mentioning explicitly all the objects in the lefthand side. This case seems the

most important in practice; however, the technique can be generalized to handle rules with more than one message in their lefthand side.

For each Maude rule, a set of Simple Maude rules is generated. First, a pattern object is chosen from the lefthand side of the Maude rule to be the locus of control. This object begins the process by locking itself to prevent the application of other rewrite rules. Locking may be implemented with a rewrite rule of type (†) that changes the shape of the object in such a way that it would fail to match with any rules other than those described below[5]. Once locked, this object sends messages to each other object appearing in the lefthand side of the instance of the original rule of type (‡). Simple Maude rules of type (†) are added for receiving these messages, locking the recipients, and responding, perhaps with some data values. Once the locus object receives positive replies from all objects named in the original rule (‡), it begins the data-matching process. The data necessary to verify a match may be sent in the locking replies already generated, or may be transferred through additional messages. Once a match is found, the locus object performs the object updates by sending messages to all altered objects, generating all new objects and new messages that the original rule specifies, sending unlocking messages to all objects in the lefthand side of the rule, and unlocking itself. These actions can easily be encoded as Simple Maude rules of type (†).

The above scheme must be extended to avoid deadlock, which may occur if two or more objects chosen as loci of control for two or more matches simultaneously attempt to lock two target objects. In such case, one of the loci objects may obtain a lock on one of the targets, and another locus object (perhaps executing a different Maude rule) may obtain a lock on the other target. Deadlock occurs since each locus object is waiting to achieve a lock on some object that is already locked by another locus object. To eliminate such deadlocks, one can implement by means of rules of type (†) a solution to the *Drinking Philosophers Problem* of Chandy and Misra [**9**] using priority and backing-out schemes. In addition to deadlock avoidance, care must be taken in the priority scheme to avoid starvation of one rule of type (‡) by other rules; for this purpose each locus object maintains priority information for each rule that applies to it, based on the number of successful complete locks achieved by the locus object for that rule. When the locus object attempts locking other objects, the corresponding priority may be communicated within the locking messages. When a locking message is received by an object already locked by some locus object with higher priority (with fewer successful complete locks), a *lock rejected* message is generated. When a locking message is received by an object already locked by some locus object with lower priority, a *cease and desist* message is sent to the lower-priority locus object. When a *lock rejected* message is received, a locus object may choose to reattempt a lock a fixed number of times or back out of the attempted rewrite altogether. When a *cease and desist* message is received, a locus object that has already obtained locks on all necessary objects may complete the rewrite, but a locus object still waiting for some subset of locks must back out of the attempted rewrite. Backing out of a rewrite may involve simply unlocking all locked objects,

[5]See the example of `Lockable` objects in [**30**, p. 232].

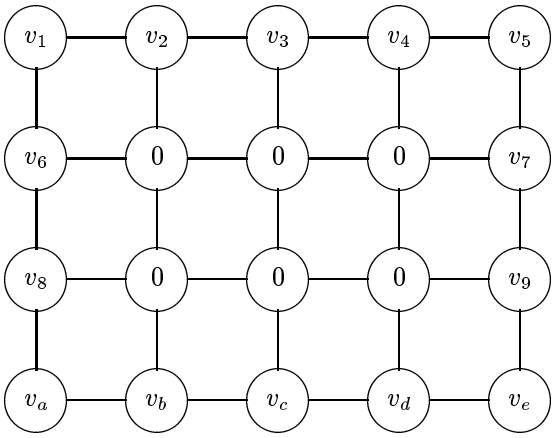

FIGURE 7. Initial state for Dirichlet problem.

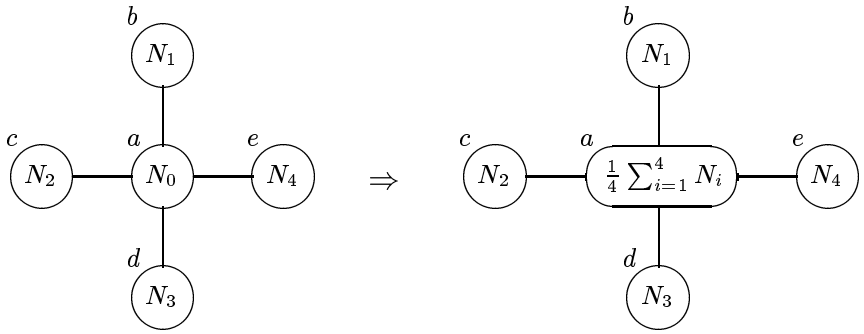

FIGURE 8. Another graph rewrite rule.

or in the case of a *cease and desist* message, sending a message to the locked object in question that atomically transfers the lock from the lower-priority locus object to the higher-priority one.

5.3. Transforming synchronous graph rewrite rules into asynchronous ones. Consider for example the Dirichlet problem from [**10**]. The problem is to find a solution to the Laplace equation $\nabla^2\Theta = 0$, where the values of Θ are fixed for the boundary cells. Initially, each interior cell has value zero, and the boundary cells have their given values v_i, as in the diagram in Figure 7.

In one parallel step of computation, the value of each interior cell is replaced by the average of the values of its four (vertical and horizontal) neighbors. This can be represented by the graph rewrite rule in Figure 8, which should be applied simultaneously everywhere, that is, in SIMD mode[6].

The computation should be performed in lock step, enforcing that all cells compute their new values simultaneously. In other words, only maximally-parallel SIMD rewrites are allowed. If one wanted to compute exactly the same

[6]The SIMD and MIMD/SIMD modes of parallel rewriting are further explained in Section 7.

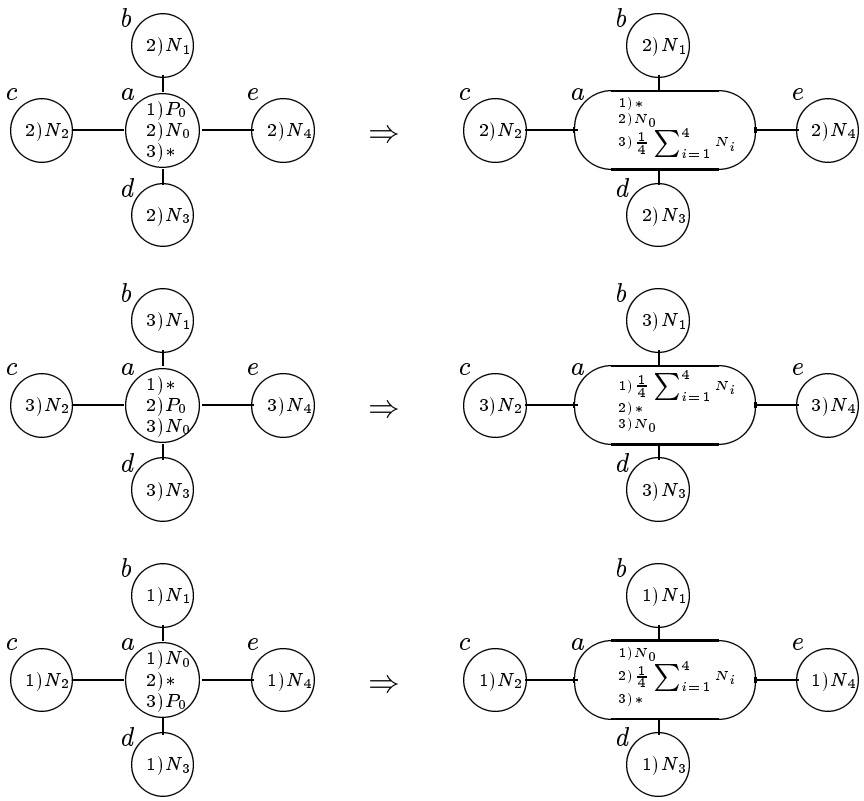

FIGURE 9. Three asynchronous graph rewrite rules.

sequence of values, but allow asynchronous computation, for example to make possible solving the problem with a MIMD/SIMD machine whose computational nodes are SIMD but operate asynchronously with decentralized controllers, one can perform the following transformation. First, where before there was only a single data value stored, we now allow three data values to be stored. Also, we introduce a distinguished data value "$*$" that cannot be matched with a number (say, by type constraints). We then produce the three rules in Figure 9, where we take the notational convenience of omitting some attributes of nodes, which are then assumed to be unchanged by the rewrite.

These three rules take the place of the one globally SIMD rule in Figure 8. The intended operation is that the symbol $*$ stands for the next value to be computed. It is an invariant that when one of these rules applies, the value of P_0 is no longer needed in the computation, and thus can be overwritten.

If one is only interested in the final result of a computation, that is, after things become stable, then, somewhat surprisingly, the original rule can also be used in an asynchronous MIMD/SIMD mode to produce the correct result. The proof that the original rule, even in MIMD/SIMD mode, leads to the same final

result as it does in SIMD mode is not difficult but beyond the capabilities of our Simple Maude compiler to determine[7], so for MIMD/SIMD machines our compiler would produce the three rule version.

In general, using the technique just illustrated, one may transform maximally-parallel SIMD rewriting into MIMD/SIMD rewriting by adding clock-time and past state attributes to all cells. Rewrite rules are then made sensitive to the clock times, enforcing that data from a consistent (artificial) clock time is used in performing all rewrites. After rewriting, a cell must retain its past state to allow its neighbors to compute their next state if they have not already done so.

Assuming that each rule changes the value in at most one node, and that the control strategy of each SIMD rule is to rewrite using that rule until quiescence, there is a useful subclass of graph rewriting for which only one past state need be recorded in the transformation described above. In order to characterize this subclass, we need to give first some definitions. Given a graph rewrite rule, we say that a node in an instance of the lefthand side of the rule is *written* if its value could be changed by application of the rule; for example, in the Dirichlet program above only the central node in each rule is written. Given a graph rewrite rule and a node A in a data graph, the *read set* of A is the set of all nodes in the graph that appear in a match of the rule where A is written; for example, the read set of a node matching node a in the Dirichlet program consists of the five nodes matching a, b, c, d, e. Also, the *write set* of A is the set of all nodes that are written in all possible matches of the rule that contain A; in the Dirichlet example, the write set of a given node A consists of itself and its four (vertical and horizontal) neighbors, because all those nodes are written when A matches the nodes a, b, c, d, e in different instances of the rule. Finally, a graph rewrite rule is *symmetric* in a data graph if the read set and the write set coincide for all nodes in the data graph. We have already shown that the Dirichlet rule is symmetric. On the other hand, a rule reading only the lefthand neighbor in a graph forming a line (e.g., a rule for *cdr*ing down a list in a list of cons cells) is not symmetric, since the righthand neighbor of each node is in the write set, but not in the read set of that node. The subclass of graph rewrite rules and data graphs for which retaining only one past state is sufficient is then defined by the following two properties:

(i) each rule is symmetric in the data graph;

(ii) for each written node each rule matches in at most one way[8];

Some graph rewrite rules can be modified in simple ways to make them conform to the above constraints. For example, some graph rewrite rules that write more than one node of a graph can be made to conform by coalescing those nodes

[7] See Section 7 for a discussion of compilation techniques for Simple Maude.

[8] For the Dirichlet example the actual rewrite rules must specify directions of the pointers; otherwise each rule could match in degenerate ways. For example, the nodes b, c, d, e could all match the same one node in the data graph, resulting in incorrect results and/or nontermination. Even if these overlaps were prevented, each rule could match at each written node in 24 different permutations. However, for the Dirichlet example the result of all of the permuted matches is the same, since addition is commutative. In general, one must mark each pointer leaving a node with direction. Simply keeping the pointers in specific registers named in the rule accomplishes this directly. One could assume that this is implicit in the graphical presentation of the rules.

into one (for the purposes of this one graph rewrite rule). Also, asymmetric rules can often be made symmetric by adding dummy backpointers to each node in the write set of the written node of the rule. These backpointers essentially perform synchronization. For example, in the *cdr*ing down a list example, the smaller end of the list may rewrite very quickly and get ahead of the head of the list in terms of rewrites. Thus a large set of past values must be kept at each node to preserve correctness. By adding backpointers that prevent rewriting anywhere more than one step ahead of the nodes in the write set, only one past value is needed at each node. Note that by recording more than one past state, one may allow some nodes in the write set to be more than one step behind. Since computation of a next value depends on the values in the read set, these nodes can never be more than one step behind. However, if additional past state values are recorded the nodes that are in the write set but not in the read set can be allowed to slip farther behind by suitable modifications to the rules.

6. Foreign interface modules

Simple Maude can support the integration within a parallel computing context of modules written in conventional languages such as Fortran and C, as well as the similar integration of entire subsystems and special-purpose hardware devices. All such programs and subsystems can be encapsulated as *foreign interface modules*. Below we briefly summarize the discussion in the paper [**31**] where this aspect of the language is treated in more detail.

The notion of foreign interface module generalizes a facility already available in Maude's functional sublanguage (OBJ) for defining *built-in sorts* and *built-in rules* [**18, 21**]. This facility has provided valuable experience with multilingual support, in this case for OBJ and Common Lisp, and can be generalized to a facility for defining foreign interface modules in Simple Maude. Such foreign interface modules have abstract interfaces that allow them to be integrated with other Simple Maude modules and to be executed concurrently with other computations; however, they are treated as "black boxes." In particular, Simple Maude's concurrent rewriting model of computation and its modular style provide a simple way of gluing a concurrent program together out of pieces that can be either written in Simple Maude or can instead be conventional programs, subsystems, or special hardware devices. Foreign interface modules may provide either a functional data type, or an object-oriented class. In the first case, the treatment will be similar to that provided in OBJ. In the second case, the abstract interface will be provided by the specification of the messages that act upon the new class of objects. This second case is also used to interface to existing systems or applications and to special-purpose hardware devices; they are treated as, possibly quite complex, black boxes.

Related efforts in multilingual support for parallel programming include: the *Linda* language developed by D. Gelernter and his collaborators at Yale [**8**], the *Strand* language designed by I. Foster and S. Taylor [**17**], the *Program Composition Notation* (PCN) designed by K. M. Chandy and S. Taylor at Caltech [**10**], and the *GLU* language developed by R. Jagannathan and A. Faustini at SRI International [**23**].

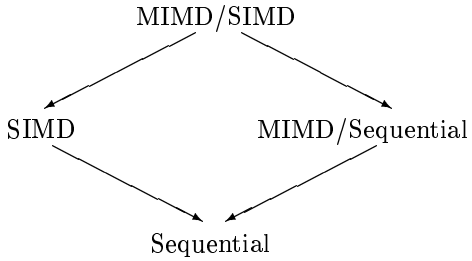

FIGURE 10. Classes of architectures.

7. Compilation onto parallel architectures

Simple Maude can be implemented on a wide variety of parallel architectures. Figure 10 shows the relationship among some general classes of architectures that we have considered. There are two orthogonal choices giving rise to four classes of machines: the processing nodes can be either a single sequential processor or a SIMD array of processors, and there can be either just a single processing node or a network of them. The arrows in the diamond denote specializations from a more general and concurrent architecture to special cases. The arrows pointing to the left correspond to specializing a network of processing nodes to the degenerate case with only one processing node; the arrows pointing to the right correspond to specializing a SIMD array to a single processor.

Each of these architectures is naturally suited to different ways of performing rewriting computations. Simple Maude has been designed so that concurrent rewriting should be relatively easy to implement efficiently in any of these four classes of machines. In the MIMD/Sequential (multiple instruction stream, multiple data) case many different rewrite rules can be applied at many different places at once, but only one rule is applied at one place in each processor. The SIMD (single instruction stream, multiple data) case corresponds to applying rewrite rules one at a time, possibly to many places in the data. The MIMD/SIMD case corresponds to applying many rules to many different places in the data, but here a single rule may be applied at many places simultaneously within a single processing node. The Rewrite Rule Machine (RRM) [**20, 3, 2, 1, 26**] is a MIMD/SIMD architecture designed with the explicit goal of supporting concurrent rewriting. Its processing nodes are two-dimensional SIMD arrays realized on a chip and the higher level structure is a network operating in MIMD mode.

The paper [**25**] gives general techniques for compiling Simple Maude onto a wide class of SIMD and MIMD/SIMD architectures, and reports on our experience in implementing those techniques in the case of the RRM. The techniques studied include:

- Top down SIMD matching and replacement.
- Program transformations taking rules for which globally SIMD lock-step execution is required into more flexible rules for which a less synchronized MIMD/SIMD regime can produce the same answers (see Section 5.3).
- Optimized mappings for fixed-topology graph rewriting applications.
- Message passing optimization.

- Efficient encoding of multiple inheritance.
- Efficient object attribute access.

We have developed a compiler for the RRM embodying many of these techniques [**25**] that achieves performance within 20% of hand coded assembly programs for the examples we have observed.

8. Concluding remarks

To realize the goal of machine-independent parallel programming in practice will require considerable effort. We view the present paper as a roadmap suggesting directions that we consider particularly promising for connecting declarative programming and parallel computing in a more intimate and widely applicable way than has been the case so far.

We have emphasized the Simple Maude language as a carefully chosen sublanguage in which three different types of rewriting—term, graph, and object-oriented rewriting—can be efficiently implemented in parallel. Since rewriting is a very simple way of implementing and parallelizing functional and other declarative and constraint-based languages (see [**33, 27**]) our approach can also be applied to their parallel implementation. In this regard, Simple Maude could be used as an intermediate language into which declarative languages are translated in order to execute them in parallel using a Simple Maude implementation. In addition, compilation techniques based on rewriting logic could be used to parallelize conventional languages by translating them into Simple Maude.

We have also emphasized the wide-spectrum character of the rewriting logic framework, which, as illustrated by the program transformation examples in Section 5, supports formal refinement of specifications into executable prototypes, and of such prototypes into efficient Simple Maude programs. Much more work remains to be done in this area, and also in the related area of program verification techniques, including techniques involving other logical formalisms such as modal or temporal logics.

To make the paradigm we have presented a reality, efficient compilation techniques applicable to wide classes of parallel machines are essential. We are encouraged by the results we have obtained so far on general compilation techniques for SIMD and MIMD/SIMD machines [**25**]; and also by the concrete experience and tools gained from implementing those techniques for the RRM. Much more work remains ahead for further developing general compilation techniques of this kind so as to cover satisfactorily most SIMD, MIMD/SIMD, and MIMD/Sequential machines, and to demonstrate the advantages and practical value of machine-independent parallel programming by means of implementations for machines spanning all these architectures.

Appendix A. Rewriting logic

This appendix gives the rules of deduction of rewriting logic.

A.1. Basic universal algebra. For the sake of simplifying the exposition, we treat the *unsorted* case; the many-sorted and order-sorted cases can be given a similar treatment. Therefore, a set Σ of function symbols is a ranked alphabet

$\Sigma = \{\Sigma_n \mid n \in \mathbb{N}\}$. A Σ-algebra is then a set A together with an assignment of a function $f_A : A^n \longrightarrow A$ for each $f \in \Sigma_n$ with $n \in \mathbb{N}$. We denote by T_Σ the Σ-algebra of ground Σ-terms, and by $T_\Sigma(X)$ the Σ-algebra of Σ-terms with variables in a set X. Similarly, given a set E of Σ-equations, $T_{\Sigma,E}$ denotes the Σ-algebra of equivalence classes of ground Σ-terms modulo the equations E ; in the same way, $T_{\Sigma,E}(X)$ denotes the Σ-algebra of equivalence classes of Σ-terms with variables in X modulo the equations E. Let $[t]_E$ or just $[t]$ denote the E-equivalence class of t.

Given a term $t \in T_\Sigma(\{x_1, \ldots, x_n\})$, and terms $u_1, \ldots, u_n$, we denote by $t(u_1/x_1, \ldots, u_n/x_n)$ the term obtained from t by *simultaneously substituting* u_i for x_i, $i = 1, \ldots, n$. To simplify notation, we denote a sequence of objects $a_1, \ldots, a_n$ by $\overline{a}$. With this notation, $t(u_1/x_1, \ldots, u_n/x_n)$ can be abbreviated to $t(\overline{u}/\overline{x})$.

A.2. The rules of rewriting logic. A *signature* in rewriting logic is a pair (Σ, E) with Σ a ranked alphabet of function symbols and E a set of Σ-equations. Rewriting will operate on equivalence classes of terms modulo the set of equations E. In this way, we free rewriting from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set E of equations is empty. The idea of rewriting in equivalence classes is well known [**22, 14**].

Given a signature (Σ, E), *sentences* of the logic are sequents of the form $[t]_E \longrightarrow [t']_E$ with t, t' Σ-terms, where t and t' may possibly involve some variables from the countably infinite set $X = \{x_1, \ldots, x_n, \ldots\}$. A *theory* in this logic, called a rewrite theory, is a slight generalization of the usual notion of theory—which is typically defined as a pair consisting of a signature and a set of sentences for it—in that, in addition, we allow rules to be labelled. This is very natural for many applications, and customary for automata—viewed as labelled transition systems—and for Petri nets, which are both particular instances of our definition.

DEFINITION A.1. *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.[9] *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*

[9] To simplify the exposition the rules of the logic are given for the case of *unconditional* rewrite rules. However, all the ideas and results presented here have been extended to conditional rules in [**28**] with very general rules of the form

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

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

the set of variables occurring in either t or t', we write $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, *or in abbreviated notation* $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$.

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sequent $[t] \longrightarrow [t']$ and write $\mathcal{R} \vdash [t] \longrightarrow [t']$ if and only if $[t] \longrightarrow [t']$ can be obtained by finite application of the following *rules of deduction*:

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

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

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

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

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

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

(iv) **Transitivity**.

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

A nice consequence of having defined rewriting logic is that concurrent rewriting, rather than emerging as an operational notion, actually *coincides* with deduction in such a logic.

DEFINITION A.2. *Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, a (Σ, E)-sequent $[t] \longrightarrow [t']$ is called a* concurrent $\mathcal{R}$-rewrite *(or just a* rewrite*) iff it can be derived from $\mathcal{R}$ by finite application of the rules 1-4.*

REFERENCES

1. H. Aida, J. Goguen, S. Leinwand, P. Lincoln, J. Meseguer, B. Taheri, and T. Winkler, *Simulation and performance estimation for the Rewrite Rule Machine*, Proc. Fourth Symp. on Frontiers of Massively Parallel Computation, McLean, Virginia, October 1992, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 336–344.
2. H. Aida, J. Goguen, and J. Meseguer, *Compiling concurrent rewriting onto the Rewrite Rule Machine*, Proc. Second Int. Workshop on Conditional and Typed Rewriting Systems, Montreal, Canada, June 1990 (S. Kaplan and M. Okada, eds.), Lecture Notes in Comput. Sci., vol. 516, Springer-Verlag, Berlin, 1991, pp. 320–332.
3. H. Aida, S. Leinwand, and J. Meseguer, *Architectural design of the Rewrite Rule Machine ensemble*, VLSI for Artificial Intelligence and Neural Networks (J. Delgado-Frias and W. R. Moore, eds.), Plenum Publ. Co., New York, NY, 1991, pp. 11–22.
4. G. Agha, *Actors*, The MIT Press, Cambridge, MA, 1986.
5. J.-P. Banâtre and D. Le Mètayer, *The Gamma model and its discipline of programming*, Sci. Comput. Programming **15** (1990), 55–77.
6. M. Bauderon and B. Courcelle, *Graph expressions and graph rewriting*, Math. Systems Theory **20** (1987), 83–127.
7. G. Berry and G. Boudol, *The Chemical Abstract Machine*, Theoret. Comput. Sci. **96** (1992), 217–248.
8. N. Carriero and D. Gelernter, *Linda in context*, Comm. Assoc. Comput. Mach. **32** (April 1989), 444–458.

9. K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, MA, 1988.
10. K. M. Chandy and S. Taylor, *An Introduction to Parallel Programming*, Jones and Bartlett Publishers, Boston, MA, 1992.
11. A. Corradini and U. Montanari, *An algebra of graphs and graph rewriting*, Category Theory and Computer Science, Paris, France, September 1991 (D. H. Pitt *et al.*, eds.), Lecture Notes in Comput. Sci., vol. 530, Springer-Verlag, Berlin, 1991, pp. 236–260.
12. D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, *Parallel programming in Split-C*, manuscript, University of California at Berkeley, 1993.
13. R. E. Cypher, J. L. C. Sanz, and L. Snyder, *Algorithms for image component labeling on SIMD mesh-connected computers*, IEEE Trans. Comput. **39** (1990), 276–281.
14. N. Dershowitz and J.-P. Jouannaud, *Rewrite systems*, Handbook of Theoretical Computer Science, Volume B (J. van Leeuwen, ed.), The MIT Press/Elsevier, Cambridge, MA, 1990, pp. 243–320.
15. H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Fourth Int. Workshop on Graph Grammars and Their Application to Computer Science, Bremen, Germany, 1990*, Lecture Notes in Comput. Sci., vol. 532, Springer-Verlag, Berlin, 1991.
16. J. H. Fasel and R. M. Keller, editors, *Proc. Workshop on Graph Reduction, Santa Fe, New Mexico, 1986*, Lecture Notes in Comput. Sci., vol. 279, Springer-Verlag, Berlin, 1987.
17. I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*, Prentice Hall, Englewood Cliffs, NJ, 1990.
18. J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler, *An introduction to OBJ3*, Proc. Int. Workshop on Conditional Term Rewriting Systems, Orsay, France, 1987 (J.-P. Jouannaud and S. Kaplan, eds.), Lecture Notes in Comput. Sci., vol. 308, Springer-Verlag, Berlin, 1988, pp. 258–263.
19. J. A. Goguen and J. Meseguer, *Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations*, Theoret. Comput. Sci. **105** (1992), 217–273.
20. J. Goguen, J. Meseguer, S. Leinwand, T. Winkler, and H. Aida, *The Rewrite Rule Machine project*, Technical Report SRI-CSL-89-6, Computer Science Laboratory, SRI International, March 1989.
21. J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, *Introducing OBJ*, Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, March 1992. To appear in: Applications of Algebraic Specification Using OBJ (J. A. Goguen, ed.), Cambridge University Press, Cambridge, UK, 1994.
22. G. Huet, *Confluent reductions: Abstract properties and applications to term rewriting systems*, J. Assoc. Comput. Mach. **27** (1980), 797–821.
23. R. Jagannathan and A. A. Faustini, *The GLU programming language*, Technical Report SRI-CSL-90-11, Computer Science Laboratory, SRI International, November 1990.
24. D. R. Jefferson, *Virtual time*, ACM Trans. Programm. Lang. Syst. **7** (1985), 404-425.
25. P. Lincoln, N. Martí-Oliet, J. Meseguer, and L. Ricciulli, *Compiling rewriting onto SIMD and MIMD/SIMD machines*, Proc. PARLE'94, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1994, to appear.
26. P. Lincoln, J. Meseguer, and L. Ricciulli, *The Rewrite Rule Machine node architecture and its performance*, Proc. CONPAR'94, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1994, to appear.
27. N. Martí-Oliet and J. Meseguer, *Rewriting logic as a logical and semantic framework*, Technical Report SRI-CSL-93-05, Computer Science Laboratory, SRI International, August 1993.
28. J. Meseguer, *Conditional rewriting logic as a unified model of concurrency*, Theoret. Comput. Sci. **96** (1992), 73–155.
29. J. Meseguer, *A logical theory of concurrent objects and its realization in the Maude language*, Research Directions in Object-Based Concurrency (G. Agha, P. Wegner, and A. Yonezawa, eds.), The MIT Press, Cambridge, MA, 1993, pp. 314–390.
30. J. Meseguer, *Solving the inheritance anomaly in concurrent object-oriented programming*,

Proc. ECOOP'93, 7th European Conf., Kaiserslautern, Germany, July 1993 (O. M. Nierstrasz, ed.), Lecture Notes in Comput. Sci., vol. 707, Springer-Verlag, Berlin, 1993, pp. 220–246.

31. J. Meseguer and T. Winkler, *Parallel programming in Maude*, Research Directions in High-Level Parallel Programming Languages (J. P. Banâtre and D. Le Métayer, eds.), Lecture Notes in Comput. Sci., vol. 574, Springer-Verlag, Berlin, 1992, pp. 253–293.
32. R. Milner, *Communication and Concurrency*, Prentice Hall International (UK), London, 1989.
33. S. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall International (UK), London, 1987.
34. W. Reisig, *Petri Nets: An Introduction*, EATCS Monogr. Theoret. Comput. Sci., vol. 4, Springer-Verlag, Berlin, 1985.
35. C. L. Seitz, J. Seizovic, and W.-K. Su, *The C programmer's abbreviated guide to multicomputer programming*, Technical Report CS-TR-88-1, California Institute of Technology, January 1988, revised April 1989.
36. M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, John Wiley and Sons, Chichester, UK, 1993.
37. C. D. Thomborson, *Does your workstation computation belong on a vector supercomputer?*, Comm. Assoc. Comput. Mach. **36** (November 1993), 41–49.
38. P. Viry, *Rewriting: An effective model of concurrency*, Proc. PARLE'94, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1994, to appear.

Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
E-mail address: {lincoln,narciso,meseguer}@csl.sri.com