REWRITING LOGIC AND MAUDE: A WIDE-SPECTRUM SEMANTIC FRAMEWORK FOR OBJECT-BASED DISTRIBUTED SYSTEMS

José Meseguer

Abstract Rewriting logic seems very well suited as a semantic framework for open object-based distributed systems. Both the distributed states and the local concurrent transitions of such systems can be naturally specified by rewrite theories in which such local concurrent transitions are described by rewrite rules. Maude is a high-performance rewriting logic language and system developed at SRI International that supports executable specification and programming, and a flexible variety of formal analyses.

As a wide-spectrum semantic framework, rewriting logic can span the gap between high-level properties and architectural designs on the one hand, and distributed or mobile system implementations on the other. Rewriting logic has been used to give a precise semantics to a number of distributed architectural notations and to obtain formal executable specifications from them. Using Maude and its associated tools, such executable specifications can then be formally analyzed in a variety of ways. Furthermore, high-level properties of such specifications expressed in nonexecutable formalisms such as temporal and modal logics can likewise be analyzed and verified. Since under quite reasonable assumptions rewriting logic specifications can be directly implemented as distributed and mobile systems, it is possible to span the gap from high-level designs to implementations without leaving the formal framework. This paper introduces rewriting logic and Maude, and surveys the experience that, thanks to the efforts of several research teams, has been gained so far in applying them to open object-based distributed systems.

1. INTRODUCTION

Rewriting logic [55] seems very well suited as a semantic framework for open object-based distributed systems. Both the distributed states and the local concurrent transitions of such systems can be naturally specified by rewrite theories in which such local concurrent transitions

are described by rewrite rules. Specifically, the distributed state can be axiomatized as an algebraic data type by an equational theory (Σ, E) , and each rewrite rule $t \longrightarrow t'$ is interpreted as a *local transition* in the distributed state of the system. That is, t and t' describe patterns for *fragments* of the distributed state of a system, and the rule specifies how a local concurrent transition can take place in such a system, changing the local state fragment from the pattern t to the pattern t' .

Rewriting logic then provides a simple inference system in which one can derive all the possible finitary concurrent transitions of the system so axiomatized. That is, concurrent computations exactly correspond to deductions in the logic. Furthermore, the *initial model* $\mathcal{T}_{\mathcal{R}}$ of a rewrite theory $\mathcal{R}$ is built out of the proof theory, so that its elements are *equivalence classes of proofs*, which are understood as *abstract, truly-concurrent computations*. Specifically, for rewrite theories specifying object-based distributed systems, Carolyn Talcott and I have shown that, under very reasonable assumptions, each equivalence class of proofs is in bijective correspondence with a *partial order of events*, in which the basic events are precisely the local concurrent rewrites [57].

An important advantage of rewriting logic specifications is that they are *executable*. Therefore, they provide an executable formal model of the specified system, that can be simulated, and can be analyzed and reasoned about using a variety of model checking and proof techniques. There are in fact three language implementations of rewriting logic, namely, ELAN [42, 7, 6], CafeOBJ [35], and Maude [16, 12]. In particular, Maude is a high-performance rewriting logic language and system developed at SRI International that supports executable specification and programming of distributed object-based systems. Furthermore, Maude is reflective, and this can be used to formally analyze the behavior of Maude specifications by flexible model checking strategies defined within the language.

Executability can be exploited not just for formal specification, symbolic simulation, and formal analysis purposes: with appropriate compilation techniques rewrite rules can in fact be the *code* of parallel, distributed, and mobile systems that are then much easier to develop, debug, and analyze than similar systems built with conventional code. This possibility has been demonstrated by the Simple Maude language and compilation techniques in the context of parallel computing [47, 48, 49]. At present, Francisco Durán, Steven Eker, Patrick Lincoln and I are working on the design and implementation of Mobile Maude [29], which will demonstrate the same possibility in the context of mobile computing and internet programming.

As a *wide-spectrum semantic framework* rewriting logic seems well suited to span the gap between high-level properties and architectural designs on the one hand, and distributed or mobile system implementations on the other. Rewriting logic has been used to give a precise semantics to a number of distributed architectural notations and concepts, and to obtain partially- or fully-defined formal executable specifications from such notations. Using Maude and its associated tools, such executable specifications can then be analyzed in a variety of ways, including symbolic simulation and debugging, and flexible forms of model checking analysis. Furthermore, using model checkig analysis and formal proofs, high-level properties of such specifications expressed in nonexecutable formalisms such as temporal and modal logics can likewise be analyzed and verified. Since, as already mentioned, rewriting logic specifications can under quite reasonable assumptions be directly implemented with competitive performance as distributed and mobile systems, it is possible to span the gap from high-level designs to distributed implementations without leaving the formal framework.

As a research program, all this is quite ambitious, and should be considered as *work in progress*, that has not yet been fully realized. However, thanks to the efforts of very gifted researchers in several research teams, a substantial body of encouraging experience already exists. The goal of this paper is to introduce rewriting logic and Maude, and to survey the experience gained so far in applying them to open object-based distributed systems. Section 2 explains how concurrent object systems are axiomatized in rewriting logic. Section 3 introduces the basic concepts of the Maude language. Section 4 explains the range of formal methods that can be used to analyze and reason about rewriting logic specifications, the formal tools available, and the specification logics in which properties can be stated. Section 5 discusses a number of applications of rewriting logic and Maude to object-based distributed systems developed so far by different researchers. The paper ends with some concluding remarks

2. CONCURRENT OBJECTS IN REWRITING LOGIC

Using an informal style of presentation, I explain how concurrent object systems are axiomatized in rewriting logic, and how concurrent object-based computations correspond to proofs in the logic and yield an algebraic model of “true concurrency”. In general, a rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, with (Σ, E) an equational specification with signature of operations Σ and a set of equational axioms E ; and with

R a collection of labelled rewrite rules. The equational specification describes the *static* structure of the system's state space as an algebraic data type. The *dynamics* of the system are described by the rules in R that specify local concurrent *transitions* that can occur in the system axiomatized by $\mathcal{R}$, and that can be applied *modulo* the equations E .

Let me explain how the state space of a concurrent object system can be axiomatized as the initial algebra of an equational theory (Σ, E) . That is, I explain the key state-building operations in Σ and the equations E that they satisfy. The concurrent state of an object-oriented system, often 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 (i.e. juxtaposition) as $_ _ : \mathbf{Conf} \times \mathbf{Conf} \rightarrow \mathbf{Conf}$. (Following the conventions of mix-fix notation, $_ _$ s are used to indicate argument positions.) The operator $_ _$ is declared to satisfy the structural laws of associativity and commutativity and to have identity $\emptyset$. Objects and messages are singleton multiset configurations, and belong to subsorts $\mathbf{Object}, \mathbf{Msg} < \mathbf{Conf}$, so that more complex configurations are generated out of them by multiset union.

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

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

where O is the object's name or identifier, C is its class, the a_i 's are the names of the object's *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. This finishes the description of some of the sorts, operators, and equations in the theory (Σ, E) axiomatizing the states of a concurrent object system. Particular systems will have additional operations and equations, specifying, for example, the data operations on attribute values. But the top level structure of the concurrent object system is typically given by the multiset union operator.

The associativity and commutativity of a configuration's multiset structure make it very fluid. We can think of it as "soup" in which objects and messages float, so that any objects and messages can at any time come together and participate in a concurrent transition corresponding to a communication event of some kind. In general, the rewrite rules in R describing the dynamics of an object-oriented system can have

the form

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

where r is the label, $\bar{x}$ is a list of the variables occurring in the rule, the M s are message expressions, $i_1, \dots, i_k$ are different numbers among the original $1, \dots, m$, and C is the rule's condition. That is, a number of objects and messages can come together and participate in a transition in which some new objects may be created, others may be destroyed, and others can change their state, and where some new messages may be created. If two or more objects appear in the lefthand side, we call the rule *synchronous*, because it forces those objects to jointly participate in the transition. If there is only one object in the lefthand side, we call the rule *asynchronous*.

For example, we can consider three classes of objects, **Buffer**, **Sender**, and **Receiver**. The buffer stores a list of numbers in its **q** attribute. Lists of numbers are built using an associative list concatenation operator, `_.` with identity `nil`, and numbers are regarded as lists of length one. The name of the object reading from the buffer is stored in its **reader** attribute. The sender and receiver objects store a number in a **cell** attribute that can also be empty (`mt`) and have also a counter (**cnt**) attribute. The sender stores also the name of the receiver in an additional attribute. The counter attribute is used to ensure that messages are received by the receiver in the same order as they are sent by the sender even though communication between the two parties is asynchronous. Each time the sender gets a new message from the buffer, it increments its counter. It uses the current value of the counter to tag the message sent to the receiver. The receiver only accepts a message whose tag is its current counter. It then increments its counter indicating that it is ready for the next message. Three typical rewrite rules for objects in these classes (where E and N range over natural numbers, L over lists of numbers, $L.E$ is a list with last element E , and `(to Z : E from (Y,N))` is a message) are

```

read(X,Y,L,E,N) : < X : Buffer | q: L . E, reader: Y >
                  < Y : Sender | cell: mt, cnt: N >
=> < X : Buffer | q: L, reader: Y >
    < Y : Sender | cell: E, cnt: N + 1 >

```

```

send(Y,Z,E,N) : < Y : Sender | cell: E, cnt: N, receiver: Z >
=> < Y : Sender | cell: mt, cnt: N > (to Z : E from (Y,N))

receive(Z,Y,E,N) : < Z : Receiver | cell: mt, cnt: N >
(to Z : E from (Y,N))
=> < Z : Receiver | cell: E, cnt: N + 1 >

```

where the **read** rule is synchronous and the **send** and **receive** rules asynchronous. These rules are applied *modulo* the associativity and commutativity of the multiset union operator, and therefore allow both object synchronization and message sending and receiving events anywhere in the configuration, regardless of the position of the objects and messages. We can then consider the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ axiomatizing the object system with these three object classes, and with R the three rules above (and perhaps other rules, such as one for the receiver to write its contents into another buffer object, that I omit).

Rewriting logic then gives an inference system to deduce, for a system axiomatized by a rewrite theory $\mathcal{R}$, which are all the finitary concurrent computations possible in such a system. Such computations are identified with *proofs* of the general form $\alpha : t \longrightarrow t'$ in the logic. In what follows, to simplify the exposition, I specialize the general inference system of [55] to the case of object systems. Idle, or “identity” computations, in which nothing changes in a state t , are denoted by t itself, and elementary rewrites corresponding to the application of a single rule are denoted by the appropriate substitution instance of the rule label; for example, **send**($b, c, 3, 1$) is the rewrite in which sender object b sends to c the value 3 with counter 1. More complex computations are then built up by parallel and sequential composition of elementary proofs, according to the following inference system, that specifies both the inferences and the new proof terms associated to proofs α and β .

- 1 **Congruence:**
$$\frac{\alpha : t \longrightarrow t' \quad \beta : u \longrightarrow u'}{\alpha \beta : t u \longrightarrow t' u'}$$
- 2 **Transitivity:**
$$\frac{\alpha : t_1 \longrightarrow t_2 \quad \beta : t_2 \longrightarrow t_3}{\alpha; \beta : t_1 \longrightarrow t_3}$$

For example, a buffer object a , and sender and receiver objects b and c can be involved in a concurrent computation in which b reads a value from a and sends it to c , and then, simultaneously, c receives it and b reads a second value from a . Suppose that we begin with the following initial configuration C_0

```

< a : Buffer | q: 7 . 9, reader: b >
< c : Receiver | cell: mt, cnt: 1 >
< b : Sender | cell: mt, cnt: 0, receiver : c >

```

Then, the above concurrent computation can be described by the proof term α , built up by repeated application of congruence and transitivity

```
(read(a,b,7,9,0) < c : Receiver | cell: mt, cnt: 1 >);
(send(b,c,9,1) < c : Receiver | cell: mt, cnt: 1 >
  < a : Buffer | q: 7, reader: b >);
(read(a,b,nil,7,1) receive(c,b,9,1))
```

and has as its final configuration C_1

```
< a : Buffer | q: nil, reader: b >
< b : Sender | cell: 7, cnt: 2, receiver : c >
< c : Receiver | cell: 9, cnt: 2 >
```

This is fine, but when do two different proofs describe the *same* concurrent computation? This typical “true concurrency” question is posed by approaches in which a more abstract description of concurrent computations is sought, and models characterizing such computations are built. For example, in our computation α we could replace the last step, namely $(\text{read}(a,b,\text{nil},7,1) \text{ receive}(c,b,9,1))$, by either of two different “interleaving” proof terms equivalent to it: one in which the *receive* happens after the *read*, and another in which they happen in opposite order.

Rewriting logic is in this sense a “true concurrency” approach. The abstract model giving a semantics to the concurrent computations of a system axiomatized by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is denoted $\mathcal{T}_{\mathcal{R}}$ and is a quotient of the algebra of proof terms modulo some simple equations that express basic equivalences between proofs. The details of this algebraic construction for an arbitrary rewrite theory can be found in [55]. In [57] Carolyn Talcott and I have given a detailed construction $\mathcal{T}_{\mathcal{R}}^o$ specializing to the case of object-oriented systems the $\mathcal{T}_{\mathcal{R}}$ model. We can give the flavor of these proof equivalences by pointing out that, firstly, all the equations E in $\mathcal{R}$ are also applied to proof terms (in particular, parallel composition of proofs in object systems is associative and commutative and has identity $\emptyset$, because multiset union enjoys those axioms) and in addition parallel and sequential composition obey, among others, the following three equations that are sufficient for proving that our proof term α above is equivalent to the two proof terms in which the last parallel step has been replaced by interleavings.

1 *Associativity.* $(\alpha; \beta); \gamma = \alpha; (\beta; \gamma).$

2 *Identities.* For each $\alpha : t \longrightarrow t'$, $\alpha; t' = \alpha$ and $t; \alpha = \alpha$.

3 *Functoriality.* $(\alpha_1; \beta_1)(\alpha_2; \beta_2) = (\alpha_1 \alpha_2); (\beta_1 \beta_2).$

By identifying proof terms using equations such as these, the model $\mathcal{T}_{\mathcal{R}}^o$ characterizes abstract concurrent computations in an object-oriented system as equivalence classes $[\alpha]$ of proof terms α . This algebraic model is constructed using initiality (see [57] for the details) and enjoys many of the good properties of algebraic constructions, including parallel and sequential composition operations. Furthermore, Carolyn Talcott and I have proved that, under reasonable assumptions about the rewrite theory $\mathcal{R}$, it is *isomorphic* to a very natural and intuitive partial order of events model $\mathcal{E}_{\mathcal{R}}$ that is fully general, in the sense of allowing both synchronous and asynchronous computations. In addition, the models $\mathcal{T}_{\mathcal{R}}^o$ and $\mathcal{E}_{\mathcal{R}}$ have a straightforward extension to models of *infinite* computations $\mathcal{T}_{\mathcal{R}}^\infty$ and $\mathcal{E}_{\mathcal{R}}^\infty$ that are again isomorphic. In fact, for rewrite theories $\mathcal{R}$ whose rules are all asynchronous and obey the actor locality laws (cf. [67]), $\mathcal{E}_{\mathcal{R}}^\infty$ specializes to the well-known partial order of events model of Hewitt and Baker for Actor systems [3, 20].

Figure 1 Equivalence between proof and event partial order descriptions of a concurrent object-oriented computation.

In the isomorphism $\mathcal{T}_{\mathcal{R}}^o \cong \mathcal{E}_{\mathcal{R}}$, an equivalence class of proof terms is mapped to a partial order of events in which the individual events are atomic rewrites. Without giving the formal details, that can be found in [57], Figure 1 illustrates this isomorphism by showing the partial order of events isomorphic to the equivalence class $[\alpha]$ of proof terms for the proof term α in our running example. In $\mathcal{T}_{\mathcal{R}}^o$ the equivalence class of proofs $[\alpha]$ labels an arrow between the beginning and ending configurations C_0 and C_1 . In $\mathcal{E}_{\mathcal{R}}$ the same arrow is instead labeled by the corresponding partial order of events. Note that the events are the elementary rewrites corresponding to applying each one of the rules, and

the order between them is the *causality* relation between such events. Thus, the first **send** and the **receive** events are causally connected, but the second **read** and the **receive** are unrelated in the causal partial order.

3. MAUDE

Maude [12, 13] is a language and system developed at SRI International whose modules are theories in rewriting logic. The most general Maude modules are called *system modules*. They have the syntax `mod  $\mathcal{R}$  endm` with $\mathcal{R}$ the rewrite theory in question, expressed with a syntax quite close to the corresponding mathematical notation¹. The equations E in the equational theory (Σ, E) underlying the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ are presented as a union $E = E' \cup A$, with A a set of *equational axioms* introduced as *attributes* of certain operators in the signature Σ —for example, an operator $+$ can be declared associative and commutative by keywords `assoc` and `comm`—and where E' is a set of equations that are assumed to be Church-Rosser and terminating *modulo* the axioms A . Furthermore $\mathcal{R}$ is assumed to be (weakly) coherent [71], which intuitively means that the equations E' and the rules R “commute with each other” modulo A in an appropriate sense. Maude supports rewriting modulo different combinations of such equational attributes A : operators can be declared with any combination of associative, commutative, and with left, right, or two-sided identity attributes.

Since we can view an equational theory as a rewrite theory whose set R of rules is empty, Maude contains a sublanguage of *functional modules* of the form `fmod  $(\Sigma, E)$  endfm`, where, as before, $E = E' \cup A$, with E' Church-Rosser and terminating modulo A . The equational logic of functional modules and of the equational part of system modules is membership equational logic [56], an expressive logic whose atomic formulas are equations $t = t'$ and membership predicates $t : s$ with s a sort. Membership equational logic is quite expressive. It supports sorts, subsorts, operator overloading, and equational partiality. But this expressiveness is achieved while being efficiently executable by rewriting and having suitable techniques for completion and for equational theorem proving [8].

Maude has a third class of modules, namely, *object-oriented modules*, that specify concurrent object-oriented systems and have syntax

¹See [12] for a detailed description of Maude’s syntax, which is quite similar to that of OBJ3 [37].

`omod  $\mathcal{R}$  endom`, with $\mathcal{R}$ a sugared form for a rewrite theory. That is, object-oriented modules are internally translated into ordinary system modules, but Maude provides a more convenient syntax for them, supporting concepts such as objects, messages, object classes, and multiple class inheritance. Some of the syntax of object-oriented modules has been illustrated in Section 2. The two key points to bear in mind about object-oriented modules are: that rewriting takes place *modulo* the axioms of associativity, commutativity, and identity axioms of the multiset union operator for configurations $_ _ : \mathbf{Conf} \times \mathbf{Conf} \longrightarrow \mathbf{Conf}$, and that rewrite rules can be either synchronous or asynchronous and have the general form explained in Section 2.

Modules in Maude have an *initial semantics*. Therefore, a system module `mod  $\mathcal{R}$  endm` specifies the initial model $\mathcal{T}_{\mathcal{R}}$ of the rewrite theory $\mathcal{R}$. Similarly, a functional module `fmod  $(\Sigma, E)$  endfm` specifies the initial algebra $T_{\Sigma, E}$ of the equational theory (Σ, E) .

In addition, like in OBJ3, Maude supports a module algebra with parameterized modules, parameter theories (with “loose” semantics) views (that is, theory interpretations) and module expressions. In Maude, this module algebra, called Full Maude, is defined inside the language by reflection, and is easily extensible with new module composition operations [30, 28]. Such a module algebra is one important concrete application of the general *metaprogramming* capabilities made possible by Maude’s support of reflection through the META-LEVEL module. Reflection, as well as the reflective capacity for defining internal strategy languages that can guide the execution of rewrite theories by means of rewrite rules at the metalevel are explained below.

Even though the Maude system is an interpreter, its use of advanced semicompilation techniques that compile each rewrite rule into matching and replacement automata [33] make it a high-performance system that can reach up to 1.66 million rewrites per second in the free theory case ($A = \emptyset$) and from 130,000 to one million rewrites per second in the associative-commutative case on a 500 MHz Alpha for some applications. In addition, a Maude compiler currently under development can reach up to 13 million rewrites per second on the same hardware for some applications.

This means that Maude can be used effectively not only for executable specification purposes, but also for declarative programming. Furthermore, as already mentioned, there is a substantial body of encouraging experience on parallel compilation for the Simple Maude sublanguage [47, 48, 49], and the design of the Mobile Maude language extension is currently under development [29] (see Section 5.3).

Executables for the Maude interpreter, a language manual, a tutorial, examples and case studies, and relevant papers can be found in <http://maude.csl.sri.com>.

Reflection and Strategies

Intuitively, a logic is reflective if it can represent its metalevel at the object level in a sound and coherent way. More precisely, Manuel Clavel and I have shown that rewriting logic is reflective [17, 11, 19] in the sense that there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that for any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) we have the following equivalence

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

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

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

Reflection is a very powerful property. It is systematically exploited in the Maude rewriting logic language implementation [12], that provides key features of the universal theory $\mathcal{U}$ in a built-in module called **META-LEVEL**. In particular, **META-LEVEL** has sorts **Term** and **Module**, so that the representations $\bar{t}$ and $\overline{\mathcal{R}}$ of a term t and a module $\mathcal{R}$ have sorts **Term** and **Module**, respectively. **META-LEVEL** has also functions **meta-reduce**($\overline{\mathcal{R}}, \bar{t}$) and **meta-apply**($\overline{\mathcal{R}}, \bar{t}, \bar{l}, \bar{\sigma}, n$) which return, respectively, the representation of the reduced form of a term t using the equations in a module $\mathcal{R}$, and the (representation of the) result of applying a rule labeled l in the module $\mathcal{R}$ to a term t at the top with the $(n + 1)$ th match consistent with the partial substitution σ . As the universal theory $\mathcal{U}$ that it implements in a built-in fashion, **META-LEVEL** can also support a reflective tower with an arbitrary number of levels of reflection.

Reflection can be exploited to define *internal rewriting strategies* [17, 18, 11], that is, strategies to guide the rewriting process whose semantics can be defined inside the logic by rewrite rules at the metalevel. In fact, there is great freedom for defining many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a fixed and closed strategy language. The idea is to use the operations **meta-reduce** and **meta-apply** as basic strategy expressions, and then to extend the module **META-LEVEL** by additional strategy expressions and corresponding semantic rules. A number of

strategy languages that have been defined following this methodology can be found in [11, 13].

4. FORMAL METHODOLOGY, TOOLS, AND SPECIFICATION LOGICS

In this section I explain how a flexible range of formal methods can be applied to rewriting logic specifications (4.1) the existing formal tools supporting these methods (4.2) and the different specification logics that can be used to state formal properties (4.3).

4.1. FORMAL METHODOLOGY

The fact that rewriting logic specifications are executable allows us to have a flexible range of *increasingly stronger formal methods*, to which a system specification can be subjected. Only after less costly and “lighter” methods have been used, leading to a better understanding and to important improvements and corrections of the original specification, is it meaningful and worthwhile to invest effort on “heavier” and costlier methods. Maude and its theorem proving tools [14, 63] (more on this below) can be used to support the following, increasingly stronger methods:

- 1 *Formal specification.* This process results in a first *formal model* of the system, in which many ambiguities and hidden assumptions present in an informal specification are clarified. A rewriting logic specification provides a formal model in exactly this sense.
- 2 *Execution of the specification.* Executable rewriting logic specifications can be used directly for simulation and debugging purposes, leading to increasingly better designs.
- 3 *Model-checking analysis.* Errors in highly distributed and non-deterministic systems not revealed by a particular execution can be found by a model-checking analysis that considers all behaviors of a system from an initial state, up to some level or condition.
- 4 *Narrowing analysis.* By using symbolic expressions with logical variables, one can carry out a symbolic model-checking analysis in which all behaviors not only from a single initial state, but also from the possibly infinite set of states described by a symbolic expression are analyzed.
- 5 *Formal Proof.* For highly critical properties it is also possible to carry out a formal proof of correctness, which can be assisted by

formal tools such as those described in Section 4.2 below. Such properties can be expressed in rewriting logic itself, or in an adequate modal or temporal logic as explained in Section 4.3

4.2. FORMAL TOOLS

The above methodology is supported by the formal tools described in what follows. First of all, Maude itself is a very versatile formal tool supporting methods 1–2 through its default interpreter, and method 3 through reflective rewriting strategies that can analyze the different concurrent computations from a given initial state checking for desired properties. In a future release of Maude we plan to support unification in a built-in way, which will make it easy to support method’s 4 narrowing analysis.

In addition to the formal methods directly supported by Maude, one can use Maude as a *formal meta-tool* [15] to build other formal tools supporting other kinds of analysis and proof. As explained in [14, 15, 54], reflection and the flexible uses of rewriting logic as a logical framework [50] are the key features making it easy to develop such formal tools and their user interfaces. Interfaces can be built using the general meta-parsing and meta-pretty printing functions of the **META-LEVEL** module, and the **LOOP-MODE** module [12], that provides a general read-eval-print loop that can be customized with appropriate rewrite rules to define the interaction with the user for each tool. Our experience suggest that it is much easier to build and maintain formal tools this way than with conventional implementation techniques. Also, because of the high performance of the Maude engine, the tools generated this way often have quite reasonable performance. Of course, special-purpose algorithms or off-the-shelf tools can also be used as components for performance or convenience reasons, since this is not excluded by Maude’s meta-tool building capabilities.

The papers [15, 63] give detailed accounts of different formal tools, including theorem proving tool, execution and analysis environments for different formalisms, a module algebra, and logic traslators, that have been defined in Maude by different authors using the above reflective methodology. Some tools are more domain-specific, or support formal analysis and reasoning in other formalisms; I discuss a few of these in Section 5. Maude-specific tools, applicable to large classes of Maude specifications, or extensions of such specifications, include the following:

An Inductive Theorem Prover. Using the reflective features found in Maude, we have built an inductive theorem prover for equational logic specifications [14] that can be used to prove induc-

tive properties of both CafeOBJ specifications [35] and of functional modules in Maude. As already mentioned, this tool can be extended with reflective reasoning principles to reason about the meta-logical properties of a logic represented in rewriting logic or, more generally, to prove metalevel properties [4].

A Church-Rosser Checker. We have additionally built a Church-Rosser checker tool [14] that analyzes equational specifications to check whether they satisfy the Church-Rosser property. This tool can be used to analyze order-sorted [36] equational specifications in CafeOBJ and in Maude. The tool outputs a collection of proof obligations that can be used to either modify the specification or to prove them. Extensions of this tool to perform equational completion and to check coherence of rewrite theories are currently under development.

Full Maude. Maude has been extended with special syntax for object-oriented specifications, and with a rich *module algebra* of parameterized modules and module composition in the Clear/OBJ style [9, 37] giving rise to the Full Maude language. All of Full Maude has been formally specified in Maude using reflection [30, 28]. This formal specification—about 7,000 lines—is in fact its implementation, which is part of the Maude distribution. Our experience in this regard is very encouraging in several respects. Firstly, because of how quickly it was possible to develop Full Maude. Secondly, because of how easy it will be to maintain it, modify it, and extend it with new features and new module operations. Thirdly, because of the competitive performance with which it can carry out complex module composition and module transformation operations, that makes the interaction with Full Maude quite reasonable.

Real-Time Maude. Based on a notion of *real-time rewrite theory* that can naturally represent many existing models of real-time and hybrid systems, and that has a straightforward translation into an ordinary rewrite theory [62], Peter Ölveczky and I are developing an execution and analysis environment for specifications of real-time and hybrid systems called Real-Time Maude [63]. This tool translates real-time rewrite theories into Maude modules and can execute and analyze such theories by means of a library of strategies that can be easily extended by the user to perform other kinds of formal analysis.

4.3. SPECIFICATION LOGICS

Rewriting logic is primarily a logic *of* change—in which the deduction directly corresponds to the change—as opposed to a logic to talk *about* change in a more indirect and global manner such as the different variants of modal and temporal logic. Such logics regard a system as a mathematical model—typically some kind of Kripke structure—about which they then make assertions about its global properties such as safety or liveness properties. Both levels of description and analysis are useful in their own right; in fact, they complement each other: one can use both logics in combination to prove system properties.

Of course, the first issue that must be addressed is the semantic integration between rewriting logic and the modal or temporal logic of choice. Usually, such an integration is straightforward, because both logics are talking about essentially the same mathematical model. In fact, the initial model $\mathcal{T}_{\mathcal{R}}$ of a rewrite theory $\mathcal{R}$ is a category with algebraic structure, where the objects correspond to system states, and the arrows correspond to concurrent system transitions. Therefore, $\mathcal{T}_{\mathcal{R}}$ can be regarded as a Kripke structure whose transitions are labeled by the arrows of the category. A variety of different modal or temporal logics can then be chosen to make assertions about such a Kripke structure, or about a closely-related structure obtained from it, such as, for example, the extension $\mathcal{T}_{\mathcal{R}}^{\infty}$ of $\mathcal{T}_{\mathcal{R}}$ to infinite computations. In fact, some of these logics can be interpreted within *first-order rewriting logic*, and we can use this first-order logic directly to express relevant properties sometimes not expressible in a given modal or temporal logic. For example, the confluence of a rewrite theory can be expressed by the first-order formula

$$\forall(x, y, z).(x \rightarrow y \wedge x \rightarrow z) \Rightarrow (\exists v.(y \rightarrow v \wedge z \rightarrow v)).$$

The point is that, when we consider arbitrary first-order formulas—and not just universally quantified conditional or unconditional rewrite rules, which are the axioms in rewrite theories—rewriting logic itself becomes another candidate for a nonexecutable specification logic.

The investigation of suitable specification logics having a good integration with rewriting logic is a current area of research. In the choice of such a specification logic there are different tradeoffs between, for example, generality, expressiveness, and amenability to different deductive and/or model checking techniques. The most general proposal advanced so far is the one by J. Fiadeiro et al. in [34], where a modal logic for reasoning about general rewrite theories is presented. But since object-oriented systems constitute a particularly wide and important application area, modal or temporal logics that provide explicit support for

object systems and can reason about their rewriting logic specifications are clearly of interest. Two candidate formalisms of this kind have been proposed. One is a version of the modal μ -calculus proposed by Ulrike Lechner [46, 44, 45] for reasoning about object-oriented Maude specifications. Another is Grit Denker's object-oriented distributed temporal logic [21], that extends the distributed object temporal logics of H.-D. Ehrich and G. Denker [32, 22, 31]. Another direction being explored is real-time formalisms: Peter Ölveczky and I are currently investigating a variant of linear time temporal logic suitable for reasoning about rewriting logic specifications of real-time systems.

One attractive prospect is the use of Maude's reflective features to build tools mechanizing the desired modal or temporal logic deduction and/or model checking, and to integrate such tools with the underlying rewriting logic and equational logic formal tools described in Section 4.2.

5. DISTRIBUTED OBJECT-BASED APPLICATIONS

I discuss work in three broad classes of applications of rewriting logic and Maude to distributed object-based systems: (i) distributed architectures and open components; (ii) networks and communication protocols; and (iii) agent and mobile languages.

5.1. DISTRIBUTED ARCHITECTURES AND OPEN COMPONENTS

It is very important to detect errors and inconsistencies as early as possible in the software design cycle. For this reason, formal approaches that can increase the analytic power of architectural notations such as architectural description languages (ADLs) and object-oriented design formalisms such as UML are very valuable. A related concern is the formal specification and analysis of *distributed component architectures*. Maude has been used by several authors in these areas to allow formal analysis of designs in early design stages, and, in some cases, to support code generation from the associated executable specifications.

- As part of a project to represent the Wright architecture description language [1] in Maude, U. Nodelman and C. Talcott have developed a representation of CSP in Maude. This is compatible with existing tools for analyzing CSP specifications, complements them by providing a rich execution environment and the ability to analyze non-finite state specifications, and provides a means of

combining CSP specifications with other notations for specifying concurrent systems.

- In [12] (Appendix E) F. Durán, C. Talcott and I present a substantial case study showing how Maude can be used to execute very high level software designs, namely architectural descriptions. We focus on a difficult case, namely, *heterogeneous* architectures, which is illustrated by a command and control example that includes dataflow, message passing, and implicit invocation subarchitectures. Using Maude, each of the different subarchitectures can not only be executed, but they can also be interoperated in the execution of the resulting overall system.
- M. Wirsing and A. Knapp [73] have stressed the practical importance of combining object-oriented design notation with formal executable specifications. Their proposal is to annotate diagrams in an existing and widely-used object-oriented notation and then derive formal executable rewriting logic specifications written in an extension of Maude. The specifications can then be used to execute and formally analyze the designs, and even to generate code in a conventional language such as Java. These ideas are further developed and implemented in Knapp's upcoming Ph.D. thesis [43].
- A related proposal is the GILO-2 scenario-based object-oriented design notation proposed by S. Nakajima and K. Futatsugi [60]. In GILO-2, scenario-based object-oriented diagrams are systematically transformed into rewriting logic formal specifications in CafeOBJ. In fact, rewriting logic provides a formal operational semantics for GILO-2 designs, and allows their execution and formal analysis.
- C. Talcott [68, 66, 69] has given a rewriting logic semantics for actor systems that are axiomatized by actor theories. She then extends actor systems by an algebra of *components*, where a component is formalized as a distributed subsystem that can include actors, messages, and other (sub-)components and that is encapsulated by a precisely defined interface that mediates its interactions with an open environment. Furthermore, she has developed methods to reason formally about such open component systems, including techniques to prove behavioral equivalence between such systems.
- G. Denker, C. Talcott and I [24] have given an executable specification of a general middleware architecture for composable distributed communication services such as fault-tolerance, security,

and so on, that can be composed and can be dynamically added to selected subsets of a distributed communications system.

- E. Najm and J.-B. Stefani [58] have used rewriting logic to give a formal semantics to the operational subset of the Reference Model for Open Distributed Processing (RM-ODP) [40]. RM-ODP specifies a language-independent computational model underlying many of the distributed computational models that have been recently proposed, including those adopted for CORBA, the Network Object system, and the Java RMI system.
- S. Nakajima [59] shows how the calculus of mobile ambients of Cardelli and Gordon [10] can be naturally expressed in rewriting logic, and then uses rewriting logic specifications in CafeOBJ to formally specify the architecture of WEB-NMS, a Java/ORB implementation of a network management system with four architectural components.

5.2. NETWORKS AND COMMUNICATION PROTOCOLS

The paper [26] surveys some of the advances made using Maude to specify and analyze communication protocols, including cryptographic protocols, and, more generally, network software such as active network programming languages, active network algorithms, network management systems, and so on. These advances can be summarized as follows:

- The paper [27] reports on joint work by researchers at Stanford and SRI with the group led by J.J. García-Luna at the Computer Communications Research Group at University of California Santa Cruz in which we used Maude very early in the design of a new reliable broadcast protocol for Active Networks. In this work, we have developed precise executable specifications of the new protocol and, by analyzing it through execution and model-checking techniques, we have found many deadlocks and inconsistencies, and have clarified incomplete or unspecified assumptions about its behavior.
- Maude has also been applied to the specification and analysis of cryptographic protocols [23], showing how Maude's reflective model-checking techniques can be used to discover attacks.
- The positive experience with security protocols has led to the adoption of Maude by J. Millen and G. Denker at SRI as the basis for

giving a formal semantics to their new secure protocol specification language CAPSL and as the meta-tool used to endow CAPSL with an execution and formal analysis environment [25].

- The paper [72] reports joint work with Y. Wang and C. Gunter at the University of Pennsylvania in using Maude to formally specify and analyze a PLAN [38] active network algorithm.
- In collaboration with M. Keaton and S. Zabele at TASC and S. Bhattacharyya at Sprint, P. Ölvecký, C. Talcott and I have specified and formally analyzed the AER/NCA [41] suite of sophisticated active network protocol components for reliable multicast. This is perhaps the most complex suite of protocols we have studied so far, because of its time-sensitive behavior and the need to analyze both correctness and performance. The Real-Time Maude tool (see Section 4.2) has allowed us to find important errors and inconsistencies in the Maude specification derived from the original use-case informal specification.
- I. Mason and C. Talcott are also using Maude to support modeling, simulation and analysis of network architectures and communication protocols taking into account resources, such as nodes, LANs and interfaces, network failures and their recovery, and composable protocols [52]. Their goal is to develop a generic infrastructure in which a variety of protocols can be modeled, simulated, and analyzed in Maude. To test their ideas, they have applied them to the Hello subprotocol of PIM-DM.
- I. Pita and N. Martí-Oliet have used the reflective features of Maude to specify the management process of broadband telecommunication networks [64, 65].
- D. Basin and G. Denker have given an in-depth experimental comparison of the advantages and disadvantages of using Maude versus using a lazy functional language like Haskell to analyze security protocols [5]. On the Maude side, this work contains valuable new optimization techniques for efficient model checking analysis of protocols.

5.3. AGENT AND MOBILE LANGUAGES

Rewriting logic and Maude have also been used to give a formal executable semantics to different agent, actor, active network, and mobile programming languages, and to develop formally-based optimization

techniques for some of these languages. Furthermore, we are currently extending Maude into the Mobile Maude language.

- As already mentioned, S. Nakajima [59] has used rewriting logic to give semantics to the calculus of mobile ambients of Cardelli and Gordon [10].
- Ishikawa et al. [39] have given a Maude specification of a representative subset of the GAEA language [61], a reflective concurrent logic programming agent language for AI applications based on a new paradigm called organic programming.
- C. Gunter, P. Kakkar, M.-O. Stehr, C. Talcott, and I are currently developing an executable formal definition of the PLAN active network programming language in Maude that can be used as a PLAN simulator (PLAN does not have a simulator at present) and as a formal tool to analyze PLAN programs.
- I. Mason and C. Talcott [51, 53, 70] use rewriting logic to give semantic definitions of actor languages, and to “compile away” certain language features by defining semantics-preserving translations between actor languages that are formalized as translations between their corresponding rewrite theories.
- van Baalen et al. [2] have used Maude to give a formal semantics to, and to analyze, key fault-tolerant protocols in the DaAgents mobile agent system. Their analysis, which includes detailed analyses of the protocols’ behavior under different fault models, has revealed mistakes and inconsistencies in the protocols’ informal specifications.
- Finally, as further explained below, we are using rewriting logic and Maude not only to specify mobile computations, but also to program them in Mobile Maude.

Mobile Maude

As already mentioned, Maude can be used not only for specifying distributed object systems, but also for programming them. At SRI, Francisco Durán, Steven Eker, Patrick Lincoln and I are currently advancing the design and implementation of Mobile Maude [29]. This is an extension of Maude supporting mobile computation that uses reflection in a systematic way to obtain a simple and general declarative mobile language design. The two key notions are *processes* and *mobile objects*. Processes are located computational environments where mobile objects

can reside. Mobile objects can move between different process in different locations, and can communicate asynchronously with each other by means of messages. Each mobile object contains its own code—that is a rewrite theory $\mathcal{R}$ —metarepresented as a term $\overline{\mathcal{R}}$, as well as its own internal state. In this way, reflection endows mobile objects with powerful “higher-order” capabilities within a simple first-order framework.

Figure 2 Object and message mobility

Figure 2 shows several processes in two locations, with (mobile) object $o3$ moving from one process to another, and with object $o1$ sending a message to $o2$.

We expect that Mobile Maude will have good support for *secure* mobile computation for two reasons. Firstly, mobile objects will communicate with each other and will move from one location to another using state-of-the-art encryption mechanisms. Secondly, because of the logical basis of Mobile Maude, we expect to be able to prove critical properties of applications developed in it with much less effort than what it would be required if the same applications were developed in a conventional language such as Java.

6. CONCLUSIONS

I have explained how rewriting logic and Maude can be used to specify, execute, and formally analyze distributed object-based systems using a flexible range of formal methods, tools, and specification logics. I have also summarized a fairly extensive body of work by different authors applying rewriting logic and Maude to distributed software architectures and components, to communication protocols and networks, and to agent and mobile languages.

This paper is a snapshot of work in progress. However, I hope to have given enough evidence to support the view that—thanks to the efforts of researchers in the US, Europe, Australia, and Japan—rewriting logic is well suited as a semantic framework for object-based distributed systems. Of course, some of the most exciting work remains ahead of us, including:

- Uses of Maude in internet programming and mobile computation, based on the upcoming release of Maude 2.0 and the Mobile Maude language currently under development.
- Further advances in property-oriented specification logics complementing rewriting logic, including development of model checking and proof techniques supported by adequate tools.
- More applications of rewriting logic and Maude to the design, analysis, and verification of secure distributed systems at all levels of abstraction: software architectures, protocols and networks, and mobile code.
- Uses of Maude as a forma meta-tool to build both a richer formal environment of tools for Maude, and a wide range of domain-specific formal tools.

Acknowledgments

I wish to thank the organizers of FMOODS 2000 for kindly suggesting the possibility of giving this talk. The work on rewriting logic and Maude is a joint research effort with all my colleagues in the Maude team, with Carolyn Talcott at Stanford University, and with other colleagues and research teams in the US, Europe, Australia, and Japan. It is therefore a pleasure for me to warmly thank all these colleagues, many of whom are mentioned by name in the body of the paper and in the references, for their many important contributions.

This work has been supported by DARPA through Rome Laboratories Contract F30602-C-0312, by DARPA and NASA through Contract NAS2-98073, by Office of Naval Research Contract N00014-99-C-0198, and by National Science Foundation Grant CCR-9900334.

References

- [1] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings 16th International Conference on Software Engineering*, 1994.
- [2] J. Van Baalen, J.L. Caldwell, and S. Mishra. Specifying and checking fault-tolerant agent-based protocols using Maude. In *Formal Approaches to Agent-Based Systems*, Lecture Notes in Computer Science, 2000. To appear.

- [3] H.G. Baker and C. Hewitt. Laws for communicating parallel processes. In *Proceedings of the 1977 IFIP Congress*, pages 987–992. IFIP Press, 1977.
- [4] D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. <http://www.informatik.uni-freiburg.de/{\homedir}basin/pubs/pubs.html>.
- [5] D. Basin and G. Denker. Maude versus haskell: an experimental comparison in security protocol analysis. Submitted for publication, May 2000.
- [6] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.
- [7] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.
- [8] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
- [9] Rod Burstall and Joseph A. Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer LNCS 86, 1980.
- [10] L. Cardelli and A.D. Gordon. Mobile ambients. Technical report, Digital Equipment Corporation, Systems Research Center, 1997.
- [11] Manuel Clavel. Reflection in general logics and in rewriting logic, with applications to the Maude language. Ph.D. Thesis, University of Navarre, 1998.
- [12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, <http://maude.csl.sri.com>.
- [13] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. A tutorial on Maude. SRI International, March 2000, <http://maude.csl.sri.com>.

- [14] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998. <http://maude.csl.sri.com>.
- [15] Manuel Clavel, Francisco Durán, Steven Eker, José Meseguer, and Mark-Oliver Stehr. Maude as a formal meta-tool. In J. Wing and J. Woodcock, editors, *FM'99 — Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer-Verlag, 1999.
- [16] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.
- [17] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.
- [18] Manuel Clavel and José Meseguer. Internal strategies in a reflective logic. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, pages 1–12, 1997.
- [19] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. Submitted for publication, 2000.
- [20] Will Clinger. Foundations of actor semantics. Technical report AI-TR-633, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1981.
- [21] G. Denker. DTL⁺: A Distributed Temporal Logic Supporting Several Communication Principles. Technical Report , SRI International, Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA 94025, 1998. *To appear*.
- [22] G. Denker and H.-D. Ehrich. Specifying Distributed Information Systems: Fundamentals of an Object-Oriented Approach Using Distributed Temporal Logic. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97), Volume 2, IFIP TC6 WG6.1 Intern. Workshop, 21-23 July, Canterbury, Kent, UK*, pages 89–104. Chapman & Hall, 1997.

- [23] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998. <http://www.cs.bell-labs.com/whonch/fmsp/index.html>.
- [24] G. Denker, J. Meseguer, and C. Talcott. Rewriting Semantics of Distributed Meta Objects and Composable Communication Services, 1999. working draft.
- [25] G. Denker and J. Millen. CAPSL Intermediate Language. In N. Heintze and E. Clarke, editors, *Workshop on Formal Methods and Security Protocols (FMSP'99), July 5, 1999, Trento, Italy (part of FLOC'99)*, 1999. <http://cm.bell-labs.com/cm/cs/whonch/fmsp99/>.
- [26] Grit Denker, José Meseguer, and Carolyn Talcott. Formal specification and analysis of active networks and communication protocols: the Maude experience. In *Proc. DARPA Information Survivability Conference and Exposition DICES 2000, Vol. 1, Hilton Head, South Carolina, January 2000*, pages 251–265. IEEE, 2000.
- [27] Denker, G. and Garcia-Luna-Aceves, J.J. and Meseguer, J. and Ölveczky, P. and Raju, J. and Smith, B. and Talcott, C. Specification and Analysis of a Reliable Broadcasting Protocol in Maude. In B. Hajek and R.S. Sreenivas, editors, *Proc. 37th Allerton Conference on Communication, Control and Computation*, 1999. [url: http://www.comm.csl.uiuc.edu/allerton](http://www.comm.csl.uiuc.edu/allerton).
- [28] Francisco Durán. A reflective module algebra with applications to the Maude language. Ph.D. Thesis, University of Málaga, 1999.
- [29] Francisco Durán, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Mobile Maude. To appear in *Proc. ASA/MA 2000 (Zurich, Switzerland, September 2000)* Springer LNCS. <http://maude.csl.sri.com>.
- [30] Francisco Durán and José Meseguer. An extensible module algebra for Maude. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, North Holland, 1998.
- [31] H.-D. Ehrich. Object Specification. Informatik-Bericht 96–07, TU Braunschweig, 1996.
- [32] H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic Publishers, 1998.
- [33] Steven Eker. Fast matching in combination of regular equational theories. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4

- of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.
- [34] J. Fiadeiro, N. Martí-Oliet, T. Maibaum, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. To appear in Proc. WADT'99, Springer LNCS, 2000.
 - [35] K. Futatsugi and R. Diaconescu. CafeOBJ report. AMAST Series, World Scientific, 1998.
 - [36] 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.
 - [37] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
 - [38] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.
 - [39] Hiroshi Ishikawa, Jose Meseguer, Takuo Watanabe, Kokichi Futatsugi, and Hideyuki Nakashima. On the semantics of GAEA—an object-oriented specification of a concurrent reflective language in rewriting logic. In *IMSA '97*, pages 70–109. Information-Technology Promotion Agency, Japan, 1997.
 - [40] ITU-T. *Recommendation X.903 — ISO/IEC International Standard 10746-3: “ODP Reference Model: Prescriptive Model”*. ISO, 1995.
 - [41] S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. Technical Report TR 99-44, University of Massachusetts, Amherst, CMPSCI, 1999.
 - [42] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In V. Saraswat and P. van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 133–160. MIT Press, 1995.
 - [43] A. Knapp. *A formal approach to object-oriented software engineering*. PhD thesis, Institut für Informatik, Universität München, 2000. To appear.
 - [44] U. Lechner. Object-oriented specification of distributed systems in the μ -calculus and Maude. In J. Meseguer, editor, *Proc.*

- First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.
- [45] U. Lechner. *Object-oriented specification of distributed systems*. PhD thesis, University of Passau, 1997.
 - [46] U. Lechner and C. Lengauer. Modal μ -Maude. In *Object Orientation with Parallelism and Persistence*, B. Freitag, C.B. Jones, C. Lengauer and H.-J. Schek, editors, Kluwer, 1996.
 - [47] 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.
 - [48] 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.
 - [49] 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.
 - [50] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.
 - [51] I. Mason and C. Talcott. A semantics preserving actor translation. In *Proc. ICALP'97*, pages 369–378. Springer LNCS 1256, 1997.
 - [52] I. Mason and C. Talcott. Simple network protocol simulation within Maude. Submitted for publication, May 2000.
 - [53] I. A. Mason and C. L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 228(1), 1999.
 - [54] José Meseguer. Rewriting logic and maude: Concepts and applications. To appear in *Proc. RTA 2000* Springer LNCS, 2000.
 - [55] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

- [56] José Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, ed., *Proc. WADT'97*, 18–61, Springer LNCS 1376, 1998.
- [57] José Meseguer and Carolyn Talcott. A partial order event model for concurrent objects. In *Proc. CONCUR'99, Eindhoven, The Netherlands, August 1999*, pages 415–430. Springer LNCS 1664, 1999.
- [58] E. Najm and J-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems, Vol. 2*, pages 157–176. Chapman & Hall, 1997.
- [59] S. Nakajima. Encoding mobility in CafeOBJ: an exercise of describing mobile code-based software architecture. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.
- [60] S. Nakajima and K. Futatsugi. An object-oriented modeling method for algebraic specifications in CafeOBJ. In *Proceedings, 19th International Conference on Software Engineering*, pages 34–44. IEEE Computer Society Press, May 1997.
- [61] Hideyuki Nakashima. Organic programming for situation-thick AI systems. In *IMSA '97*, pages 156–163. Information-Technology Promotion Agency, Japan, 1997.
- [62] Peter Csaba Ölveczky and José Meseguer. Specification of real-time and hybrid systems in rewriting logic. Submitted for publication, 1999.
- [63] Peter Csaba Ölveczky and José Meseguer. Real-Time Maude: a tool for simulating and analyzing real-time and hybrid systems. Submitted for publication, May 2000.
- [64] Isabel Pita and Narciso Martí-Oliet. A Maude specification of an object oriented database model for telecommunication networks. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.
- [65] Isabel Pita and Narciso Martí-Oliet. Using reflection to specify transaction sequences in rewriting logic. In J.L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques*, volume 1589 of *Lecture Notes in Computer Science*, pages 261–276. Springer-Verlag, 1999.
- [66] C. L. Talcott. An actor rewrite theory. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Else-

- vier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.
- [67] C. L. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, number 4 in *Electronic Notes in Theoretical Computer Science*. North Holland, 1996.
- [68] C. L. Talcott. Interaction semantics for components of distributed systems. In E. Najm and J-B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 154–169. Chapman & Hall, 1997.
- [69] C. L. Talcott. Actor theories in rewriting logic, 1999. Submitted for publication.
- [70] C. L. Talcott. Towards a toolkit for actor system specification. In *AMAST 2000*, *Lecture Notes in Computer Science*, 2000. To appear.
- [71] 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.
- [72] Bow-Yaw Wang, José Meseguer, and Carl A. Gunter. Specification and formal analysis of a PLAN algorithm in Maude. To appear in *Proc. DSVV 2000*.
- [73] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm>.