1. Introduction

The theory and applications of rewriting logic have been vigorously developed by researchers all over the world during the past eleven years. The attached bibliography includes more than three hundred papers related to rewriting logic that have been published so far. Three international workshops on rewriting logic have been held in the United States, France, and Japan [222,172,139], and a fourth will be held in Italy in 2002. Furthermore, as explained later in this roadmap, several language implementations and a variety of formal tools have also been developed and have been used in a wide range of applications.

Several snapshots of the state of rewriting logic research — some more global in scope, and others restricted to specific areas such as concurrency or object-based systems — have appeared so far [223,227,229,228]. The present survey is another such snapshot, but it is restricted on purpose on two counts: first in its length, which is relatively short; and second in discussing only work within the rewriting logic area. In particular, no attempt has been made to discuss work on related approaches serving as logical or semantic frameworks. In fact, it is not even a detailed survey of work in rewriting logic; instead, as its name suggests, it is a roadmap to help somebody interested in this area get the lay of the land, that is, a first general overview of the main concepts, results, and applications in what we think is a promising research area. In particular, the references cited in the roadmap do not try to be exhaustive, but only...
to give some illustrative examples. However, the bibliography itself contains all the relevant references that we are aware of at this time.

2. Basic concepts

In rewriting logic [218] the basic axioms are rewrite rules of the form $t \rightarrow t'$, with $t$ and $t'$ expressions in a given language. There are two complementary readings of a rewrite rule $t \rightarrow t'$, one computational, and another logical:

- **computationally**, the rewrite rule $t \rightarrow t'$ is interpreted as a local transition in a concurrent system; that is, $t$ and $t'$ describe patterns for fragments of the distributed state of a system, and the rule explains how a local concurrent transition can take place in such a system, changing the local state fragment from an instance of the pattern $t$ to the corresponding instance of the pattern $t'$.

- **logically**, the rewrite rule $t \rightarrow t'$ is interpreted as an inference rule, so that we can infer formulas of the form $t'$ from formulas of the form $t$.

The computational and logical viewpoints are not exclusive: they complement each other and are, in some sense, in the eyes of the beholder. For example, a simple rewrite theory whose rewrite rules rewrite ground multisets built out of some constants by means of an associative and commutative multiset union operator, denoted, say, by $\odot$, has an obvious computational reading as a (place/transition) Petri net; and an equally obvious logical reading as a tensor theory in propositional linear logic (for a discussion of these two readings see [211]).

A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma; E; L; R)$, where $(\Sigma; E)$ is the equational theory modulo which we rewrite, $L$ is a set of labels, and $R$ is a set of labeled rules.\footnote{For simplicity we will assume that $R$ consists of unconditional labeled rules of the form $l : t \rightarrow t'$, but all we say extends naturally to conditional rules that may contain rewrites in their conditions [218].} In the case of a Petri net, $\Sigma$ consists of the binary multiset union operator $\odot$ and one constant for each place in the net, $E$ consists of the associativity and commutativity equations for multiset union, $L$ is the set of labels of the net’s transitions, and $R$ is the set of transitions. Since we rewrite modulo the equations $E$, what are really rewritten are equivalence classes of terms modulo $E$. In the Petri net example this corresponds to the fact that each transition rewrites (a fragment of) the current multiset of places (graphically depicted as a “marking”, with as many “tokens” in a place as its multiplicity) modulo the associativity and commutativity of multiset union.

As a consequence, the relevant sentences — that may or may not be provable by the above theory $\mathcal{R}$ — are sequents of the form $[t]_E \rightarrow [t']_E$, where $t$ and $t'$ are $\Sigma$-terms, possibly involving some variables, and $[t]_E$ denotes the equivalence class of the term $t$ modulo the equations $E$. The provable sentences are exactly those derivable by the following inference rules:\footnote{For simplicity, we treat here unsorted (and unconditional) rewriting logic; but the logic is in fact parameterized by the choice of its underlying equational logic: unsorted, many-sorted, order-sorted, membership, etc.}

1. **Reflexivity.** For each $[t] \in T_{\Sigma; E}(X)$, $[t] \rightarrow [t]$. 
(2) **Congruence.** For each \( f \in \Sigma_n, n \in \mathbb{N}, \)

\[
[t_1] \rightarrow [t'_1] \ldots [t_n] \rightarrow [t'_n] \\
[f(t_1, \ldots, t_n)] \rightarrow [f(t'_1, \ldots, t'_n)].
\]

(3) **Replacement.** For each rule \( l : [t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)] \) in \( R, \)

\[
[w_1] \rightarrow [w'_1] \ldots [w_n] \rightarrow [w'_n] \\
[l(w/x)] \rightarrow [l'(w'/x)].
\]

(4) **Transitivity.**

\[
[t_1] \rightarrow [t_2] [t_2] \rightarrow [t_3] \\
[t_1] \rightarrow [t_3].
\]

Computationally, the provable sequents describe all the complex concurrent transitions of the system axiomatized by \( \mathcal{R}. \) Logically, they describe all the possible complex deductions from one formula to another in the logic axiomatized by \( \mathcal{R}. \)

Besides having an inference system, rewriting logic also has a **model theory** with natural computational and logical interpretations. Furthermore, each rewrite theory \( \mathcal{R} \) has an **initial model** \( \mathcal{I}_\mathcal{R} \) [218]. The idea is that we can decorate the provable sequents with **proof terms**, indicating how indeed they have been proved. Computationally, a proof term is a description of a, possibly complex, concurrent computation; logically, it is of course a description of a logical deduction. The question is then, when should two such proof terms be considered **equivalent** descriptions of the same computation/deduction?

In the model \( \mathcal{I}_\mathcal{R} \) this is answered by equating proof terms according to natural equivalence equations [218]. In this way, we obtain a model \( \mathcal{I}_\mathcal{R} \) with a category structure, where the objects are \( E \)-equivalence classes of ground \( \Sigma \)-terms, and the arrows are equivalence classes of proof terms. Identities are naturally associated with reflexivity proofs; and arrow compositions correspond to transitivity proofs. The computational and logical interpretations are then obvious, since a category is a structured transition system; and logical systems have been understood as categories since the early work of Lambek on deductive systems. The proof theory and model theory of rewriting logic are related by a **completeness theorem**, stating that a sequent is provable from \( \mathcal{R} \) if and only if it is satisfied in all models of \( \mathcal{R} \) [218].

Yet another very important property of rewriting logic is **reflection** [67,65,70]. Intuitively, a logic is reflective if it can represent its metalevel at the object level in a sound and coherent way. Specifically, rewriting logic can represent its own theories and their deductions by having 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

\[
\mathcal{R} \vdash t \rightarrow t' \iff \mathcal{U} \vdash \langle \mathcal{R}, \tilde{t} \rangle \rightarrow \langle \mathcal{R}, \tilde{t}' \rangle,
\]

where \( \tilde{t} \) and \( \tilde{t}' \) are terms, of respective sorts \( \text{Module} \) and \( \text{Term} \), 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 [65,71].
Reflection is a very powerful property: it allows defining rewriting strategies by means of metalevel theories that extend \( \mathcal{U} \) and guide the application of the rules in a given object-level theory \( \mathcal{R} \) [65]; it can be efficiently supported in a language implementation by means of descent functions [71]; it can be used to build a variety of theorem proving and theory transformation tools [65,82,83]; it can endow a rewriting logic language with powerful theory composition operations [120,116,118,124]; and it can be used to prove metalogical properties about families of theories in rewriting logic, and about other logics represented in the rewriting logic logical framework [12,84].

How should rewrite theories be executed in practice? First of all, in a general rewrite theory \( \mathcal{R} = (\Sigma, E, L, R) \) the equations \( E \) can be arbitrary, and therefore, \( E \)-equality may be undecidable. Assuming that the equations \( E \) are unconditional, a general solution is to transform \( \mathcal{R} \) into a rewrite theory \( \mathcal{R}^E = (\Sigma, \emptyset, L \cup L_E, R \cup E \cup E^{-1}) \) in which we view the equations \( E \) as rules from left to right \( (E) \) and from right to left \( (E^{-1}) \), labeled by appropriate new labels \( L_E \). In this way, we can reduce the problem of rewriting modulo \( E \) to the problem of standard rewriting, since we have the equivalence

\[
\mathcal{R} \vdash [t] \rightarrow [t'] \iff \mathcal{R}^E \vdash t \rightarrow t'.
\]

In actual specification and programming practice we can do much better than this, because the equational theory \( (\Sigma, E) \) is typically decidable. A commonly occurring form for the decidable equational theory \( (\Sigma, E) \) is with \( E = E' \cup A \), where \( A \) is a set of equational axioms for which we have a matching algorithm, and \( E' \) is a set of Church–Rosser and terminating equations modulo \( A \). In these circumstances, a very attractive possibility is to transform \( \mathcal{R} = (\Sigma, E' \cup A, L, R) \) into the theory \( \mathcal{R}^A = (\Sigma, A, L \cup L_{E'}, R \cup E') \). That is, we now view the equations \( E' \) as rules added to \( R \), labeled with appropriate new labels \( L_{E'} \). In this way, we reduce the problem of rewriting modulo \( E \) to the much simpler problem of rewriting modulo \( A \), for which, by assumption, we have a matching algorithm. The question is, of course, under which conditions is this transformation complete, that is, under which conditions do we have an equivalence

\[
\mathcal{R} \vdash [t]_E \rightarrow [t']_E \iff \mathcal{R}^A \vdash [t]_A \rightarrow [t']_A.
\]

Conditions guaranteeing this equivalence center around different variations on the notion of coherence, which is a form of “relative confluence” between equations and rules. Methods for checking coherence, or for achieving it by a process of “relative completion”, have been proposed by Viry in several papers [314,315,318].

Even when the rewrite theory is coherent and the language implementation supports rewriting modulo \( A \), executing rewrite theories is nontrivial, because the rules \( R \) in general are neither Church–Rosser nor terminating. Furthermore, some rules in \( R \) may have additional variables on their right-hand sides, yet another source of nondeterminism. For this reason, sequential implementations of rewriting logic typically support rewriting strategies that let the user specify how the rules should be applied [183,67,23,68,17,319,65]. Such strategies can be defined in metalevel theories by reflection, as already indicated, or they may be part of a strategy language supported by a language implementation. However, one should not forget that rewriting logic is an intrinsically concurrent formalism, that can be used directly for concurrent and
distributed programming (see for example \[235,203,128\]). Therefore, whereas in a sequential implementation we are *simulating* a concurrent execution, and need a strategy to choose a particular interleaving computation, in a truly concurrent execution non-determinism is a fact of life, and we may care much less about how rules are applied, and be much less able to control their application in practice. We may in fact allow many different computations, while still imposing some weaker requirements such as different forms of fairness.

3. Rewriting logic and formal methods

The fact that, under reasonable assumptions, 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, it is meaningful and worthwhile to invest effort on “heavier” and costlier methods. A rewriting logic language implementation, together with an associated environment of formal tools, can be used to support the following, increasingly stronger methods \[79\]: (1) formal specification, (2) execution of the specification, (3) model-checking analysis, (4) narrowing analysis, and (5) formal proof.

Executability, combined with program transformation and compilation techniques, has yet another key advantage, namely, that rewriting logic specifications validated by the above formal methods can then be directly transformed and compiled for efficient execution. In fact, the state-of-the-art in rewriting logic language implementations (see Section 6) suggests that for many applications the implementations thus obtained, besides being correct by construction, can compete in efficiency with implementations developed in conventional languages.

The above methodology should be supported by formal tools. First of all, a reflective rewriting logic implementation can directly support methods 1–3, and can also be used as a *reflective metatool* to develop other formal tools for methods 3–5. Maude has been used in exactly this way \[82,83,262\] to build tools such as an inductive theorem prover; a tool to check the Church–Rosser property, coherence, and termination, and to perform Knuth–Bendix completion; and a tool to specify, analyze and model check real-time specifications \[265,262\]. Some of the above tools have also been integrated within the formal tool environment of CafeOBJ \[143\]. Similarly, as further discussed in Section 5, both ELAN and Maude have been used to develop a wide variety of formal tools and automated deduction algorithms, based on quite different logics.

Rewriting logic is primarily a logic *of* change in which the deduction directly corresponds to the change \[211\], 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.
The integration of these two logical levels is usually straightforward, because both logics are talking about essentially the same mathematical model. In fact, the initial model $\mathcal{T}_R$ of a rewrite theory $R$ is a category with algebraic structure, where the objects correspond to system states, and the arrows correspond to concurrent system transitions. Therefore, $\mathcal{T}_R$ 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 $\mathcal{T}_R^\infty$ of $\mathcal{T}_R$ to infinite computations.

The investigation of suitable specification logics having a good integration with rewriting logic is an active 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. Two general proposals for modal logics for reasoning about general rewrite theories are those of Faddeiro et al. in [136], and the coalgebraic approach of Pattinson [270]. 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 $\mu$-calculus proposed by Lechner for reasoning about object-oriented Maude specifications [194,195,198], and another is Denker’s object-oriented distributed temporal logic [90]. A direction recently explored by Ölveczky and supported by the model-checking features of the Real-Time Maude tool [265] is a timed linear time temporal logic suitable for reasoning about rewriting logic specifications of real-time systems [262]; in a similar vein, Beffara et al. have used rewrite rules and ELAN strategies to verify properties of timed automata [14]. An even more recent direction actively pursued at SRI is the development of an explicit state model checker to check linear temporal logic formulas on the general class of rewriting logic specifications executable in Maude; this model checker will be part of the upcoming Maude 2.0 distribution.

4. Semantic framework applications

The computational and logical interpretations of rewriting logic lead to applications that use it: as a semantic framework, in which different languages and models of computation are expressed; or as a logical framework, in which different logics and inference systems are likewise expressed [208]. We first discuss semantic framework applications.

4.1. Models of computation

This section presents concrete evidence (in highly condensed form; see [223,227] for much more detailed discussions) for the thesis that a wide variety of models of computation, including concurrent ones, can be naturally and directly expressed
as rewrite theories in rewriting logic. As a consequence, models hitherto quite different from each other can be naturally unified and interrelated within a common framework.

The following models of computation have been naturally expressed in rewriting logic: (1) equational programming, which is the special case of rewrite theories whose set of rules is empty and whose equations are Church–Rosser, possibly modulo some axioms $A$; (2) lambda calculi and combinatory reduction systems [218,192,193,293,290]; (3) labeled transition systems [218]; (4) grammars and string-rewriting systems [218]; (5) Petri nets, including place/transition nets, contextual nets, algebraic nets, colored nets, and timed Petri nets [218,223,291,295,266,287]; (6) Gamma and the Chemical Abstract Machine [218]; (7) CCS and LOTOS [237,208,314,45,89,311,309,201]; (8) the $\pi$ calculus [316,290]; (9) concurrent objects and actors [218,220,300,302,304]; (10) the UNITY language [218]; (11) concurrent graph rewriting [223]; (12) dataflow [223]; (13) neural networks [223]; (14) real-time systems, including timed automata, timed transition systems, hybrid automata, and timed Petri nets [266,262]; and (15) the tile logic [146,147,135] model of synchronized concurrent computation [231,40,34,148].

Since the above specifications of models of computation as rewrite theories are typically executable, this suggests that rewriting logic is a very flexible operational semantic framework to specify the semantics of such models. What is not immediately apparent from the above list is that it is also a flexible mathematical semantic framework at the level of concurrency models. That is, quite often a well-known mathematical model of concurrency happens to be isomorphic to the initial model $T_R$ of the rewrite theory $R$ axiomatizing that particular model, or at least closely related to such an initial model. Some examples will illustrate this point: (1) in [193] it is shown that for rewrite theories of the form $R = (\Sigma, \emptyset, L, R)$, with the rules $R$ left-linear, $T_R$ is isomorphic to a model based on residuals and permutation equivalence proposed by Boudol; (2) the same paper also shows that for $R = (\Sigma, E, L, R)$ a rewrite theory axiomatizing an orthogonal combinatory reduction system, including the $\lambda$-calculus, a quotient of $T_R$ by a few additional equations is isomorphic to a well-known model of parallel reductions based on residuals and permutation equivalence; (3) the paper [295] shows in detail that for $R = (\Sigma, E, L, R)$ a rewrite theory axiomatizing a place/transition net, $T_R$ is naturally isomorphic (in the categorical sense) to the Best-Devillers net process model — a result essentially known from the coincidence of $T_R$ with the Meseguer–Montanari algebraic model of nets [218] and the Degano–Meseguer–Montanari algebraic characterization of net processes — and then generalizes this natural isomorphism to one between $T_R$ and a Best-Devillers-like model for $R$ the axiomatization of an algebraic net; (4) the papers [45,89] show that for $R = (\Sigma, E, L, R)$ a rewrite theory axiomatizing CCS, a truly concurrent semantics causal model based on proved transition systems is isomorphic to a quotient of $T_R$ by a few additional axioms; (5) the paper [237] shows that for $R = (\Sigma, E, L, R)$ a rewrite theory axiomatizing a concurrent object-oriented system satisfying reasonable requirements, a subcategory of $T_R$ is isomorphic to a partial order of events model which, for asynchronous object systems corresponding to actors, coincides with the finitary part of the Hewitt–Baker partial order of events model.
An important additional development in this area is the $\rho$-calculus of Cirstea and Kirchner [57,54,59,60]. This is a very general rewrite theory that can play for rewriting logic specifications a role similar to that played by the $\lambda$-calculus in functional computing; its generality is shown by the fact that $\rho$-terms generalize the rewriting logic proof terms defined in [218]. Furthermore, the $\rho$-calculus can simulate the $\lambda$-calculus itself. In fact, by replacing and generalizing the $\lambda$-calculus idea of function application by that of rule application, the $\rho$-calculus unifies both the $\lambda$-calculus and first-order rewriting. In analogy with $\lambda$-calculi, there are typed versions, including a simply typed $\rho$-calculus and a “$\rho$ cube” [58,62].

4.2. Semantics of programming languages

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

The use of rewrite rules to define the semantics of programming languages is of course not new. In a higher-order version it goes back to the use of semantic equations in denotational semantics; in a first-order version, the power of equational specifications to give semantic definitions of conventional languages has been understood and used for a long time. However, both the lambda calculus and executable equational specifications implicitly assume that such language definitions can be given in terms of functions, and rely on the Church–Rosser property to reach the result of an execution. For sequential languages, by making the state of the computation explicit, a functional description of this kind can always be achieved. The situation becomes more difficult for languages that support highly concurrent and nondeterministic applications, and where the possibly nonterminating interactions between processes or components — as opposed to the computation of an output value from given inputs — are often the whole point of a program. Such languages and applications do not have a natural equational description in terms of functions, but do have a very natural rewriting logic semantics, not only operationally (by means of rewriting steps) but also denotationally ($\mathcal{T}_\#$ and related models).

Since structural operational semantics definitions can be used for languages not amenable to a functional description, it is natural to compare them with rewriting logic definitions. Their relationship has been discussed in detail in [208]. In fact, both “big-step” and “small-step” structural operational semantics definitions can be naturally regarded as special formats of corresponding rewrite theory definitions [208]. Tile models provide yet another systematic way of understanding structural operational semantics definitions as tile rewrite theories [146–148], which can then be mapped into rewriting logic for execution purposes [231,40,34]. There is also a close connection between rewriting logic and Mosses’s modular structural operational semantics (MSOS) which has been recognized from the beginning [247,248], and that has led to ongoing work on a Maude Action Tool to execute MSOS definitions and Action Semantics definitions [32].
A number of encouraging case studies giving rewriting logic definitions of programming languages have already been carried out by different authors. Firstly, some of the models of computation discussed in Section 4.1 are so closely connected with languages that their rewriting logic specifications are also language specifications. Good examples are rewriting logic definitions of the lambda calculus and (mini-) ML, CCS, the π-calculus, and sketches of UNITY and Gamma, which are given in some of the references cited in Section 4.1. Secondly, the usefulness of rewriting strategies to specify program evaluations has been clearly demonstrated in ELAN specifications, for example of Prolog and of the functional-logic programming language Babel [320], and also in the Maude executable specifications for CCS developed by Bruni and Clavel [63,34], and by Verdejo and Martí-Oliet [311,310]. Thirdly, the fact that rewriting logic naturally supports concurrent objects has proved very useful in formally specifying a number of novel concurrent and mobile languages. For example, Ishikawa et al. have given a Maude specification of a representative subset of GAEA, a reflective concurrent logic programming language developed at ETL, Japan [164,163]. Mason and Talcott have used 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 [212]. Van Baalen, Caldwell, and Mishra have used Maude to give a formal semantics to the DaAgent mobile agent system and to analyze key fault-tolerant protocols in that language [9]; their analysis has revealed mistakes and inconsistencies in the protocols’ informal specifications. Yet another example is the formal executable specification in Maude of UPenn’s PLAN active network programming language [236,322]. Maude itself has been used to define the semantics of its Mobile Maude extension [120]. Finally, Maude has been used not only to specify programming languages, but also to specify and verify microprocessors in work by Harman [154,155].

4.3. Distributed architectures and 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 like UML are quite valuable. A related concern is the formal specification and analysis of distributed component architectures.

Rewriting logic has been used by several authors in these areas to allow formal analysis of software designs and, in some cases, to support code generation from the associated executable specifications. Relevant work in this direction includes: (1) work of Nodelman and Talcott representing both the Wright architecture description language and its underlying CSP semantics in Maude; (2) work of Durán, Meseguer, and Talcott on semantic interoperation of heterogeneous software architectures based on their rewriting logic semantics [232] (see also Appendix E of [74]); (3) work of Wirsing and Knapp on the systematic transformation of UML diagrams and similar object-oriented notations into formal executable rewriting logic specifications in Maude, which can
then be used to execute and formally analyze the designs, and even to generate code in a conventional language such as Java \[326,185,186,327\]; (4) work by Fernández and Toval formalizing in Maude the UML metamodel and its evolution \[305,132\], with applications to formal analysis and prototyping \[131,306\]; (5) work by Nakajima and Futatsugi on the transformation of GILO-2 scenario-based object-oriented design diagrams for execution and formal analysis \[254\]; (6) work by Talcott on a rewriting logic semantics for actor systems axiomatized by actor theories \[298–302\]; such systems can be extended by an algebra of components, that are encapsulated by interfaces, and that can include actors, messages, and other (sub-)components; in addition Talcott has developed methods to reason formally about such open component systems; (7) work by Denker, Meseguer, and Talcott on 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 \[100\]; (8) work by Najm and Stefani giving a rewriting logic semantics to the operational subset of the Reference Model for Open Distributed Processing (RM-ODP) \[249–251\] (see also \[127\]); (9) work by Nakajima that uses rewriting logic specifications in CafeOBJ to formally specify the architecture of WEB-NMS, a Java/ORB implementation of a network management system \[252\]; and (10) work by Albarrán, Durán, and Vallecillo on interoperating Maude executable specifications with distributed component platforms such as CORBA and SOAP \[1–3\].

4.4. Specification and analysis of communication protocols

Because of its flexibility to model distributed objects with different modes of communication and interaction, rewriting logic is very well suited to specify and analyze communication protocols, including cryptographic protocols, and, more generally, network software such as active network programming languages, active network algorithms, and network management systems.

Applications of this kind include: (1) work by researchers at Stanford, SRI, and at the Computer Communications Research Group at University of California Santa Cruz using Maude to analyze the early design of a new reliable broadcast protocol for active networks \[96,97\]; (2) work of Denker, Meseguer, and Talcott on the specification and analysis of cryptographic protocols using Maude \[98,99\] (see also \[277\]); (3) work of Basin and Denker on an experimental comparison of the advantages and disadvantages of using Maude versus using Haskell to analyze security protocols \[10\]; (4) work of Millen and Denker at SRI using Maude to give a formal semantics to their new cryptographic protocol specification language CAPSL, and to endow CAPSL with an execution and formal analysis environment \[92–95\]; (5) work of Wang, Gunter, and Meseguer using Maude to formally specify and analyze a PLAN active network algorithm \[322\]; (6) work by Ölvecky et al. using Real-Time Maude to specify and analyze the AER/NCA suite of active network protocol components for reliable multicast \[268\]; (7) work of Verdejo, Pita, and Martí-Oliet on the Maude specification and verification of the FireWire leader election protocol \[312\]; (8) work of Mason and Talcott on modeling, simulation and analysis of network architectures and communication protocols \[213\]; and (9) work of Pita and Martí-Oliet using the reflective features of
Maude to specify some management processes of broadband telecommunication networks [271–273].

5. Logical framework applications

Rewriting logic is like a coin with two inseparable sides: one computational and another logical. The generality and expressiveness of rewriting logic as a semantic framework for concurrent computation has also a logical counterpart. Indeed, rewriting logic is also a promising logical framework in which many different logics and formal systems can be naturally represented and interrelated [208,209]. Using a rewriting logic implementation such representations can then be used to generate a wide range of formal tools.

5.1. Representing, mapping, and reasoning about logics

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

$$\Phi: \mathcal{L} \rightarrow RWLogic.$$ 

The map $\Phi$ should be conservative, that is, it should preserve and reflect theoremhood. The reason why rewriting logic is a good framework is that the formulas of a logic $\mathcal{L}$ can typically be equationally axiomatized by an equational theory, and the rules of inference can then be typically understood as rewrite rules, that may be conditional if the inference rules have “side conditions”. Therefore, the mappings $\Phi$ are usually very simple and direct. Furthermore, using reflection we can define and execute a map $\Phi$ of this kind inside rewriting logic itself by means of an equationally defined map

$$\bar{\Phi}: \text{Module}_{\mathcal{L}} \rightarrow \text{Module}.$$ 

The map $\bar{\Phi}$ can be defined by extending the universal theory $\mathcal{U}$, which has a sort $\text{Module}$ representing rewrite theories (see Section 2), with the equational definition of a new sort $\text{Module}_{\mathcal{L}}$ whose terms represent (finitely presentable) theories in the logic $\mathcal{L}$.

In fact, we can go a step further, and represent inside rewriting logic a mapping $\Theta: \mathcal{L} \rightarrow \mathcal{L}'$ between any two finitary logics $\mathcal{L}$ and $\mathcal{L}'$ as an equationally defined function $\bar{\Theta}: \text{Module}_{\mathcal{L}} \rightarrow \text{Module}_{\mathcal{L}'}$. If the map $\Theta$ is computable, then, by a metatheorem of Bergstra and Tucker it is possible to define the function $\bar{\Theta}$ by means of a finite set of Church–Rosser and terminating equations. That is, such functions can be effectively defined and executed within rewriting logic.

In summary, using reflection, mappings between logics, including maps representing other logics in rewriting logic, can be internalized and executed within rewriting logic,
There is yet another reason why rewriting logic is very useful for logical framework applications. Thanks to reflection and the existence of initial models, rewriting logic can not only be used as a logical framework in which the deduction of a logic $L'$ can be faithfully simulated, but also as a metalogical framework in which we can reason about the metalogical properties of a logic $L$. Basin, Clavel, and Meseguer have begun studying the use of reflection, induction, and Maude’s inductive theorem prover enriched with reflective reasoning principles to prove such metalogical properties [11–13].

A good number of examples of representations of logics in rewriting logic have been given by different authors, often in the form of executable specifications, including: (1) the logics represented by Martí-Oliet and Meseguer in [208,209], including equational logic, Horn logic with equality, linear logic, logics with quantifiers, and any sequent calculus presentation of a logic for a very general notion of “sequent”; (2) the map $\text{LinLogic} \rightarrow \text{RWLogic}$ in [208,209] representing propositional linear logic was subsequently specified in a reflective way in Maude by Clavel and Martí-Oliet [63,65]; (3) the map $\text{HOL} \rightarrow \text{Nuprl}$ between the logics of the HOL and Nuprl theorem provers has been specified in Maude by Stehr, Naumov, and Meseguer [257,296]; (4) Dowek, Hardin, and Kirchner have presented (what obviously are) rewrite theories for doing deduction modulo an equational theory of equivalence between formulas specified by the equations $E$ of the rewriting logic axiomatization, both for first-order and higher-order logics [109–111]; (5) the connections with rewriting logic of that work have been made explicit by Viry, who has given a coherent sequent calculus rewrite theory in this style in [317,318] (see also [101]); (6) Stehr and Meseguer have defined a natural representation map $\text{PTS} \rightarrow \text{RWLogic}$ of pure type systems (a parametric family of higher-order logics generalizing the $\lambda$-cube) in rewriting logic [293]; and (7) Bruni, Meseguer, and Montanari have defined a mapping $\text{Tile Logic} \rightarrow \text{RWLogic}$ from tile logic into rewriting logic that can be used to execute tile logic specifications [34,37–40].
5.2. Specifying and building formal tools

Theorem provers and other formal tools have underlying *inference systems* that can be naturally specified and prototyped in rewriting logic. Furthermore, the strategy aspects of such tools and inference systems can then be specified by rewriting strategies. The researchers in the ELAN group have developed an impressive collection of rewriting logic specifications for different automated deduction inference systems, including the already-mentioned theorem proving modulo methods [109–111], logical languages, unification and narrowing [183,320], Knuth–Bendix completion with constraints [173], higher-order unification [15], combination of unification algorithms [275], constraint solving [46–50], and termination and tree-automata techniques [149,150]. In a somewhat similar vein, the work of Schorlemmer explores the relationships between rewriting logic and Levy and Agusti’s general bi-rewriting approach to automated deduction [282–285].

In Maude, formal tools have typically a reflective design that, by metarepresenting theories as data, easily allows inference steps that may transform the object theory. Strategies are then rewrite theories controlling the application of such metalevel inference rules at the meta-meta-level. We have already mentioned in Section 3 several such tools that are part of the Maude formal environment, namely, an inductive theorem prover; Church–Rosser, coherence, and termination checkers, and a Knuth–Bendix completion tool [80–83,117,119,123]; plus the Real-Time Maude tool [265,262,267]. Also closely related to Maude itself is the Full Maude tool, which extends Maude with special syntax for object-oriented specifications, and with a rich *module algebra* of parameterized modules and module composition operations [120,116,126]. This method of building formal tools is not restricted to Maude-related tools: One can generate tools from their rewriting logic specifications for any finitary logic, such as: (1) a proof assistant built by Stehr for the Open Calculus of Constructions, which extends Coquand and Huet’s calculus of constructions with equational reasoning and a flexible universe hierarchy [292]; (2) the Maude Action Tool [32] already mentioned in Section 4.2; (3) a CCS execution and verification environment developed by Verdejo and Martí-Oliet [311,310]; (4) a tool by Havelund and Roşu for testing linear temporal logic formulae on finite execution traces [157–160,278]; and (5) a tool by Fischer and Roşu to automatically check an abstract interpretation against user-given properties [137].

6. Language implementations

Several language implementation efforts in France, Japan, and the US have adopted rewriting logic as their semantic basis and support executable rewriting logic specification and programming.

The ELAN language has been developed at LORIA (CNRS, INRIA, and Universities of Nancy) [183,320,26–28,20]. It has as modules *computational systems*, consisting of a rewrite theory and a *strategy* to guide the rewriting process [23,219,17,29]. As already discussed in Section 5, this group and their collaborators have developed a very impressive collection of examples and case studies in areas such as logic programming languages, constraint solving, higher-order unification, equational theorem-proving, and
other such computational systems. Besides the ELAN interpreter, there is also a high-
performance ELAN compiler, including compilation of AC-rewriting [243–246,176].

The CafeOBJ language implementation, developed at the Japan Advanced Institute
of Science and Technology (JAIST) in Kanazawa [141,140,104,105,108], contains OBJ
as its functional sublanguage, and supports object-oriented specifications. Furthermore,
its semantics is multi-logical and includes hidden-sorted versions of equational and
rewriting logic [102–105]. The CafeOBJ language has been the basis of an ambitious
research effort — the Cafe Project — involving several research institutions in Japan,
Europe and the US, as well as several Japanese industries, to promote formal methods
applications in software engineering [138,143]. This project has achieved a distributable
version of the language and further work on its semantics, a collection of specification
libraries and case studies, an environment, and a collection of theorem proving tools
supporting different forms of verification. Furthermore, a compiler has been developed
in addition to the Cafe interpreter implementation [260,165].

The Maude language has been developed at SRI, in Menlo Park, California
[220,85,64,79,76]. The equational logic underlying Maude’s rewriting logic is mem-
bership equational logic [226,30,31], and gives rise to a sublanguage of functional
modules. System modules specify general rewrite theories, and object-oriented mod-
ules provide syntactic sugar for object-oriented rewrite theories. These modules can be
combined by module composition operations supported by Full Maude [116,126,121].
Maude’s high-performance rewrite engine makes extensive use of advanced semicom-
pilation techniques; there is also a high-performance experimental Maude compiler. In
addition, Maude efficiently supports reflection through its METALeVEL module [71,79].
Maude has been used in a wide range of applications, many of which have been dis-
cussed in this paper.

Acknowledgements

We would like to thank all our colleagues who have given us comments on earlier
versions of this roadmap and have helped us in compiling the following bibliography;
we thank specially Miguel Palomino, whose extensive help has made possible the
existence of the bibliography.

References

[1] A. Albarrán, F. Durán, A. Vallecillo, From Maude specifications to SOAP distributed implementations:
a smooth transition, In: Proc. VI Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2001,
specifications, in: Proc. 6th ECOOP Internat. Workshop on Component-Oriented Programming,
extended Maude, in: K. Futatsugi, J.A. Goguen, J. Meseguer (Eds.), OBJ/CafeOBJ/Maude Workshop


[280] M. Sánchez, J.L. Herrero, J.M. Murillo, J. Hernández, Guaranteeing coherent software systems when composing coordinated components, in: A. Porto, G.-C. Roman (Eds.), Coordination Languages and


