From OBJ to Maude and Beyond

José Meseguer

University of Illinois at Urbana-Champaign, USA

Dedicated to Joseph Goguen on his 65^{th} *Birthday*

Abstract. The OBJ algebraic specification language and its Eqlog and FOOPS multiparadigm extensions are revisited from the perspective of the Maude language design. A common thread is the quest for ever more expressive computational logics, on which executable formal specifications of increasingly broader classes of systems can be based. Several recent extensions, beyond Maude itself, are also discussed.

1 Introduction

Joseph and I met for the first time in San Francisco on February 25, 1977 at the First (and last!) International Symposium on *Category Theory Applied to Computation and Control* [80]. We wrote our first paper together in 1977 [57]. We worked very closely together at SRI from 1980 to 1988, when the bulk of our joint published work appeared, and, after his departure to Oxford and his subsequent return to San Diego, we have continued collaborating in various ways. In honoring him as a friend, colleague, and mentor of those early years, I want to reflect on some great things we did together at SRI from the perspective of how they have influenced the work that other colleagues and I have done on Maude in the 1990s and in the present decade. Since Maude itself is evolving and expanding in different directions, my reflections, will not only look at the past, but will also try to sketch what those directions, leading beyond Maude itself, look like. My views are necessarily subjective and partial, and my memory too; but that does not prevent me from trying to recollect things as best as I can, and from taking full responsibility for my own words and actions.

One common thread of our joint work at SRI was the OBJ language. Joseph and I worked on OBJ1 with David Plaisted [63], and then, in the *annus mirabilis* 1983–84, with Kokichi Futatsugi and Jean-Pierre Jouannaud on OBJ2 [47]. Then came OBJ3 [53], the most ambitious and far-reaching language design and implementation on which we worked with Claude and Hélène Kirchner, Patrick Lincoln, Aristide Mégrelis, and Timothy Winkler. A long paper combining in some way the OBJ2 and OBJ3 ideas appeared later [65], within an entire book dedicated to the OBJ experience [66]. I try to explain in this paper how not only OBJ, but also the Eqlog [59] and FOOPS [61] multiparadigm extensions of OBJ, on which Joseph and I also worked together at SRI, have influenced Maude. But to make better sense of all this, I think that it may be worthwhile to first present my own perspective on the specification language design *challenges* that we have

K. Futatsugi et al. (Eds.): Goguen Festschrift, LNCS 4060, pp. 252–280, 2006.

been trying to meet all along, and which have motivated the design of each of these languages.

1.1 System Specification Vs. Property Specification

In discussing different uses of logic in computer science, considerable confusion can arise from lack of relevant distinctions. One that I have repeatedly found useful to clarify some key issues is the distinction between *system specification* and *property specification*. In a system specification we are after an unambiguous specification of a given system and how it actually *works*. In its most useful form, a system specification is *executable* and therefore provides an *executable model* of the system. Such specifications are enormously useful, since a system design can then be *tested* and *analyzed* in various ways, and it is possible to *refine*, sometimes even automatically, such an executable model into an actual system implementation.

By contrast, when specifying *properties* of a system we are not necessarily after an executable model of our system. Instead, we *assume it*, as either already given or to be developed later, and specify such properties in a typically nonexecutable manner: for example in first-order logic, higher-order logic, or some temporal logic. That is, the properties we specify have an *intended model*, namely the system design captured by a system specification, and we are interested in *verifying* by different methods that the intended model *satisfies* the properties stated in our property specification.

1.2 System Specification in Computational Logics

The above distinction brings us to the heart of a real problem: how can we *formally*, that is using logical and mathematical methods, verify a property if the system specification we have is *informal*, that is, if it does not precisely define a *mathematical model* of our system? This is indeed a genuine problem. Having a formal grammar is a necessary but insufficient condition: we also need a formal *semantics*. This is where the rub comes with system specifications based on conventional programming languages. For some such languages nobody has managed so far to give a complete formal semantics and therefore the only unambiguous "specifications" of some languages are their different compilers, which may exhibit different behaviors. Here is where computational logics can render an invaluable service. A computational logic can either:

1. be used as a *declarative programming language* with a *precise mathematical semantics* to *directly* express system specifications; or
2. be used to *give a precise mathematical semantics* to a conventional programming language, so that a system specified by a program in such a language will *indirectly* acquire a precise mathematical meaning in the computational logic.

I have not yet defined what I mean by a *computational logic*. The simplest practical answer is: a logic that you can implement as a programming language. That is, you can define and implement a programming language whose programs are exactly *theories* in the given logic and whose program execution is *logical deduction*. You then call such a language a *declarative* programming language. The point is that from the earliest times of computability theory, logical formalisms and mathematical definitions of computability have gone hand in hand. For example, Herbrand-Gödel computable functions are defined by *equational* theories; and Church computability is defined in terms of the lambda calculus. Over time, this has given rise to various declarative programming languages. For example, pure Prolog is a declarative programming language associated to Horn logic; pure ML and Haskell are declarative programming languages associated to the typed lambda calculus; OBJ is a declarative programming language based on order-sorted equational logic; and Maude is a declarative programming language based on rewriting logic.

One can always blur the above distinctions, but this is not very helpful. For example, there is always the Quixotic and amusing possibility of declaring that *everything is a logic!*, including, say, C++, thus arriving at a toothless notion of "logic". The opportunities for confusion and obscurantism are indeed endless; but such verbal games are for the most part a waste of time. Furthermore, it is possible to give *meta-logical* requirements for declarative programming languages that cut through silly verbal games of this kind: Joseph and I gave such requirements in terms of institutions in [60]; and I gave more detailed requirements in terms of general logics in [85].

1.3 The Quest for More Expressive Computational Logics

A lot of water has gone under the bridges since the 1930s. Founding computation on a theory of recursive functions was a great achievement at its time and is still very useful today; but it is clearly a limited theory, and we know it. There is, for example, no meaningful way of thinking of internet computations as definable by recursive functions. Massive changes in the nature of computing and emergence of entirely new applications do not make older computational logics and declarative languages incorrect or useless; but they can make them limited, relegated to specific *niches*. If a wider, more general applicability beyond such niches is desired, computational logics are typically in need of either generalization or replacement. One good example is functional programming, which is of course a very elegant and powerful way of programming *functional* applications. It is certainly possible to add bells and whistles to a functional language, for example by grafting a process calculus on top of it, so as to make it suitable for nonfunctional applications such as distributed computing. But what is the *logic* of such a centaur? The fact that it can be given a semantics, just as Java can, proves nothing, since the real issue is whether the resulting language remains declarative in the precise sense of programs being theories in a logic, for a decent meta-theoretic notion of logic, and computation being deduction in such a logic.

Therefore, to preserve the declarative nature of a language, when extending it to cover new application areas, one should think primarily of how its underlying logic can be extended, and only secondarily about the extended syntax: declarative language design is primarily a task of *logic design*. The design space is therefore the space, in fact the *category*, of logics. But there are tight design constraints and tradeoffs that require good judgment. Not all logics are computational; and having a recursively enumerable set of deducible formulas is *not* a sufficient condition: first-order logic has that, but it is hopeless as a programming language. The logic has to remain *lean and mean* in order to allow efficient implementations as a programming language, and not just as a theorem prover. Yet, the whole point of an extension is to make the logic more expressive. How to achieve both goals in an optimal way is the challenge.

OBJ and its extensions are a good case in point. As algebraic specification/equational programming languages, OBJ2 [47] and OBJ3 [53,65] were arguably the most expressive such languages in the 1980s. But they were, by the very nature of their underlying order-sorted equational logic [62] and their associated operational semantics [52,70], *functional languages*. Extending OBJ in a *multiparadigm* way was a task that Joseph and I undertook in the mid 1980s, resulting in two new language designs: Eqlog [59], and FOOPS [61]. Eqlog unified functional/equational programming and Horn-logic programming; its logic design task was to embed order-sorted equational logic and Horn logic without equality into a suitable Horn logic with equality [60]. FOOPS unified equational/functional programming, Horn-logic programming, and object-oriented programming. Although an underlying model-theoretic semantics was given, based on algebraic data types with hidden sorts and behavioral equivalence between them in the sense of [58,94], FOOPS fell short of having an underlying logic with modules as theories and computation as deduction. This was remedied later, by theoretical developments presenting various proposals for a hidden or "observational" equational logic [50,51,56,55,122,64,68,11,10,9,115,116,117,30,120]. In hindsight, one can view CafeOBJ [46], BOBJ [54] and BMaude [96] as full-blooded declarative languages that achieve in a more satisfactory way many of the FOOPS goals.

1.4 Rewriting Logic and Maude

With rewriting logic [87,88,13] and Maude [86,90,18,19], several of us undertook the task of unifying within a single declarative language: (i) equational/functional programming; (ii) object-oriented programming; and (iii) concurrent/distributed programming. That (iv) Horn-logic programming was also naturally embeddable in this framework was clear from the early stages of this project [89,90], but at the operational semantics level this required a generalization of narrowing that was achieved later [132,133]. Three more insights emerged over time as part of different research collaborations: (v) that real-time and hybrid systems could be naturally specified in rewriting logic [108]; (vi) that higher-order type theory was naturally embeddable in rewriting logic [130]; and (vii) that probabilistic

systems were likewise expressible in a natural probabilistic extension of rewriting logic and could be simulated within rewriting logic itself [73,4]. In spite of being multiparadigm in all the above (i)–(vii) ways, rewriting logic remains remarkably lean and mean: it is a very simple formalism and, thanks to Steven Eker, has a very high-performance Maude implementation. Modules are indeed theories in the logic, and nothing more. Computation *is* deduction, and theories have initial models [88,13], which give semantics to modules and support inductive reasoning. Furthermore, operational properties such as termination can be usefully formulated and verified by adopting this logical/deductive viewpoint [36,79]

2 From Order-Sorted to Membership Equational Logic

Rewriting logic contains membership equational logic [92] as a sublogic. In Maude's language design this is reflected in its sublanguage of *functional modules*, for equational theories with initial semantics, and of *functional theories* for equational theories with "loose" semantics. Therefore, in relating OBJ and Maude the first task at hand is relating their corresponding equational logics.

One key reason why OBJ2 and OBJ3 were so expressive was their order-sorted type theory. That one should use types to make any reasonable sense of algebraic specifications goes without saying. But the problem with many-sorted equational logic is that it does not deal well with partiality. Many simple operations, such as selectors in data structures or just simple arithmetic operations, are *partial*. To the embarrassment of many-sorted specifications, simple trade examples, such as the perennial stacks or the rational numbers, cannot be given elegant many-sorted specifications: the top of the empty stack or division by zero raise their ugly heads and require ugly ad-hoc solutions.

The appeal of order-sorted equational logic [62] is that, by allowing the expressive power of subtypes (subsorts), many partial functions become total on appropriate subsorts. Furthermore, function symbols can be subsort overloaded, which is very convenient in practice. But there are limits to the kind of partiality expressible by typing means alone, which are those available in order-sorted algebra. When the definedness of a function depends on *semantic conditions* such as, for example, the fact that for the concatenation of two paths in a graph to be defined the target node of the first must coincide with the source node of the second, order-sorted equational logic is not enough. This was understood early on, and led to formulating notions of unconditional [49] or conditional [52] *sort constraints*; but how to extend order-sorted equational logic so as to fully account for conditional sort constraints remained an open question.

The appeal of membership equational logic (MEL) [92] is that it gives a full account of partiality, and even a systematic, functorial way of relating partial and total specifications [92,95]. Furthermore, as shown in [92], it embeds in a conservative way the "right" version of order-sorted equational logic, one that solves several anomalies, including the lack of pushouts of theory morphisms, in the version given in [62]. But does membership equational logic remain lean

and mean? The relevant facts are that it: (i) has a well-developed operational semantics by rewriting (see the systematic study [12], which also deals with many other automated deduction techniques); (ii) enjoys a high-performance Maude implementation; (iii) is a quite simple logic; and (iv) has initial and free models [92], on which inductive proof methods and inductive theorem proving tools can be based [12,20]. From these facts it seems fair to conclude that the answer is definitely yes.

In summary, therefore, we can view OBJ3 as a *sublanguage* of Maude's functional sublanguage. The generalization from OBJ3 to Maude is further stressed by the fact that Maude supports order-sorted notation as convenient syntactic sugar for membership equational logic axioms. In membership equational logic atomic propositions are either equations $t = t'$, or memberships $t : s$, stating that term t has sort s. A subsort declaration $s < s'$ is then just syntactic sugar for a conditional axiom $x : s \Rightarrow x : s'$. Similarly, an order-sorted operator declaration $f : s_1 \dots s_n \longrightarrow s$ is syntactic sugar for the conditional axiom $x_1 : s_1 \wedge \dots \wedge x_n : s_n \Rightarrow f(x_1, \dots, x_n) : s$.

A membership equational theory is a pair (Σ, H) with Σ a signature specifying the kinds, sorts, and function symbols, and with H a set of Horn clauses involving both equations and memberships. Kinds classify potentially meaningful expressions, and sorts within a kind classify actually defined expressions. Terms having a kind but not a sort correspond to undefined or error expressions. For example, $2/0$ is in the *Number* kind but has no sort. For execution purposes we typically impose some requirements on such a theory. First of all, its Horn clauses H may be decomposed as a union $E \cup A$, with A a set of equations that we will reason *modulo* (for example, A may include associativity, commutativity and/or identity axioms for some of the operators in Σ). Second, the remaining Horn clauses E are typically required to be Church-Rosser[1] modulo A, so that we can use the conditional equations in E as equational rewrite rules modulo A. Third, for some applications it is useful to make the equational rewriting relation[2] *context-sensitive* [76,77]. This can be accomplished by specifying a function $\mu : \Sigma \longrightarrow \mathbb{N}^*$ assigning to each function symbol $f \in \Sigma$ (with, say, n arguments) a list $\mu(f) = i_1 \dots i_k$ of *argument positions* , with $1 \leq i_j \leq n$, which must be fully evaluated (up to the context-sensitive equational reduction strategy specified by μ) in the order specified by the list $i_1 \dots i_k$ before applying any equations whose lefthand sides have f as their top symbol. For example, for $f = \mathit{if\_then\_else\_fi}$ we may give $\mu(f) = \{1\}$, meaning that the first argument must be fully evaluated before the equations for *if_then_else_fi* are

[1] See [12] for a detailed study of equational rewriting concepts and proof techniques for MEL theories.

[2] As we shall see, in a rewrite theory $\mathcal{R}$ rewriting can happen at two levels: (1) *equational rewriting* with (possibly conditional) equations E; and (2) *non-equational rewriting* with (possibly conditional) rewrite rules R. These two kinds of rewriting *are different*. Therefore, to avoid confusion I will qualify rewriting with equations as *equational rewriting*.

applied[3]. Therefore, for execution purposes we can specify a membership equational theory as a triple $(\Sigma, E \cup A, \mu)$, with A the axioms we rewrite modulo, and with μ the map specifying the context-sensitive equational reduction strategy. A Maude functional module is then, essentially, a specification of the form `fmod` $(\Sigma, E \cup A, \mu)$ `endfm`.

3 Rewriting Logic: From OBJ to Maude

As already mentioned, the whole point of rewriting logic [87,88,13] and Maude [86,90,18,19] was to unify within a single logic and associated declarative language: (i) equational/functional programming; (ii) object-oriented programming; and (iii) concurrent/distributed programming. For this unification, a purely equational/functional framework would be clearly unsuitable[4] The challenge therefore was to find a lean and mean superlogic of equational logic in which this unification could take place.

A related challenge was to make some sense of the quite diverse menagerie of concurrency models that were around, often competing with each other as the "right" model of concurrency. A key strategy in this competition game was to produce, sometimes quite complicated, translations from other models, adduced as proof of universality of the proposed model. Implicit in this strategy was the belief that, given enough time, the right model, capable of expressing all the relevant concurrency concepts would emerge. This search for the Holy Grail of concurrency is certainly a chivalrous one; but I find serious grounds for being skeptical about its success. The main difficulty is that concurrency encompasses a very wide range of phenomena: there are concurrent functional programs, concurrent grammars, dataflow networks, actors, Petri nets of various ilks and colors, various synchronous and asynchronous process calculi, neural networks, and so on. Although translations between some of these models are possible, the fact that in this way some concurrency features can simulate others, perhaps in a complex way, is not particularly helpful.

In my view, what was missing was a *computational logic for concurrency* that could serve as a *semantic framework* in which different concurrency models could

[3] As in OBJ2–3, in Maude maps μ specifying context-sensitive equational reduction strategies are called *evaluation strategies* [47,40,18], and $\mu(f) = i_1 \ldots i_k$ is specified with the `strat` keyword followed by the string $(i_1 \ \ldots \ i_k \ 0)$, with 0 indicating evaluation at the top of the function symbol f. For an in-depth study of the relationship between OBJ/Maude evaluation strategies and context-sensitive rewriting see [76,77].

[4] The key point is that concurrency and nondeterminism cannot be *directly modeled* in an equational/functional framework, which typically assumes determinism in the form of a Church-Rosser property. Therefore, one needs special devices to model some concurrency aspects *indirectly*. Two good examples of indirectly modeling concurrency within a purely functional framework are the ACL2 semantics of the JVM using a scheduler [101], and the use of lazy data structures in Haskell to analyze cryptographic protocols [7].

be naturally unified without requiring any translations. That is, in a logic one can define quite different *theories* which have associated *models*. The logic then allows one to understand in a unified way all such models as models in the same logic; but there is plenty of room for diversity between them. Furthermore, once we understand that a logical framework of this kind can give us an enormous range of possibilities for naturally expressing different concurrency phenomena, we realize that we can have a general *framework* without in any way needing a general *model*, whatever that means.

Is rewriting logic a suitable general framework in exactly this sense? The answer is necessarily an empirical one, and can never be claimed to be definitive. But the amount of positive evidence gathered up to now, thanks to the research of different people and covering indeed a very wide range of concurrency models, is in my view very strong. The key point is the naturalness and directness with which different concurrency models can be expressed as rewrite theories. It is not a matter of complicated *encodings*: typically the original representations of a model and those of its associated rewrite theory are isomorphic. Since all this is a matter carefully documented in many papers and in several rewriting logic surveys, I will not go over the, indeed quite large, body of work backing the view that rewriting logic is a very expressive general framework for concurrency. I refer the reader to the survey paper [82]; and for an explanation of how rewriting logic unifies and improves upon other semantic frameworks such as algebraic semantics and structural operational semantics (SOS) to the more recent papers [97,98].

3.1 Rewrite Theories: Their Execution and Formal Analysis

A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$, with: (1) $(\Sigma, E \cup A, \mu)$ a membership equational theory with "modulo" axioms A and context-sensitive equational reduction strategy μ; (2) R a set of *labeled conditional rewrite rules* of the general form

$$r : (\forall X)\ t \longrightarrow t' \ \text{ if } \ (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \longrightarrow w_l') \qquad (1)$$

where the variables appearing in all terms are among those in X, terms in each rewrite or equation have the same kind, and in each membership $v_j : s_j$ the term v_j has kind $[s_j]$; and (3) $\phi : \Sigma \longrightarrow \mathcal{P}(\mathbb{N})$ a mapping assigning to each function symbol $f \in \Sigma$ (with, say, n arguments) a set $\phi(f) = \{i_1, \ldots, i_k\}$, $1 \leq i_1 < \ldots < i_k \leq n$ of *frozen argument positions*[5] under which it is forbidden to perform any rewrites.

Intuitively, $\mathcal{R}$ specifies a *concurrent system*, whose states are elements of the initial algebra $T_{\Sigma/E \cup A}$ specified by $(\Sigma, E \cup A)$, and whose *concurrent transitions* are specified by the rules R, subject to the frozenness requirements imposed by ϕ.

[5] In Maude, $\phi(f) = \{i_1, \ldots, i_k\}$ is specified by declaring f with the `frozen` attribute, followed by the string $(i_1 \ \ldots \ i_k)$. Although originated by a quite different motivation, frozen operators have some similarities with notions such as "non-coherent operators" in CafeOBJ [46], and "non-congruent" operators in BOBJ [54].

The frozenness information is important in practice to forbid certain rewritings. For example, when defining the rewriting semantics of a process calculus, one may wish to require that in prefix expressions $\alpha.P$ the operator $\_.\_$ is *frozen in the second argument*, that is, $\phi(\_.\_) = \{2\}$, so that P cannot be rewritten under a prefix. Note that a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, \phi, R)$ specifies two kinds of *context-sensitive* rewriting requirements: (1) equational rewriting with E modulo A is made context-sensitive by μ; and (2) non-equational rewriting with R is made context-sensitive by ϕ. But the maps μ and ϕ impose *different types* of context-sensitive requirements: (1) $\mu(f)$ specifies a list of arguments where we *are allowed* to rewrite with equations in E; and (2) $\phi(f)$ specifies arguments where we *are forbidden* to rewrite with the rules R. The maps μ and ϕ substantially increase the expressive power of rewriting logic, because various order-of-evaluation and context-sensitive requirements, which would have to be specified by explicit rules in a formalism like SOS, become implicit and are encapsulated in μ and ϕ.

For execution purposes a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ should satisfy some basic requirements that are assumed to hold for Maude *system modules*. Such modules are specifications of the form `mod` $(\Sigma, E \cup A, \mu, R, \phi)$ `endm`. First, in the membership equational theory $(\Sigma, E \cup A, \mu)$, E should be ground Church-Rosser modulo A – for A a set of equational axioms for which matching modulo A is decidable – and ground terminating modulo A, up to the context-sensitive strategy μ[6]. Second, the rules R should be *coherent* with E modulo A [136]; intuitively, this means that, to get the effect of rewriting in equivalence classes modulo $E \cup A$, we can always first simplify a term with the equations in E to its canonical form modulo A, and then rewrite with a rule in R. Finally, the rules in R should be *admissible* [18], meaning that in a conditional rewrite rule of the form (1), besides the variables appearing in t there can be extra variables in t', provided that they also appear in the condition and that they can all be *incrementally instantiated* by either matching a pattern in a "matching equation" or performing breadth first search in a rewrite condition (see [18] for a detailed description of admissible equations and rules).

Computation in Maude is then deduction with the inference rules of rewriting logic (see [13]) that are efficiently implemented by the Maude engine under the above executability assumptions. Specifically, equivalence classes modulo $E \cup A$ are represented by their unique canonical forms modulo A. That is, Maude performs equational rewriting to reach a canonical form with the equations in E modulo A by means of the `reduce` command. This is entirely analogous to OBJ's `reduce` command for equational specifications, but applies now to more general theories. It also supports two variants of fair rewriting with the rules R modulo A which, in combination with equational rewriting and under the coherence assumption, achieves the effect of rewriting with R in $(E \cup A)$-equivalence classes. These two commands are the rule-fair `rewrite` command; and the rule and po-

[6] μ-termination is a weaker requirement than termination [77]; the interactions between context-sensitive rewriting and the Church-Rosser property are somewhat subtle [75,78].

sition fair `frewrite` command which, for object-based systems (see Section 3.3) is also object and message fair. Furthermore, the context-sensitive requirements provided by μ and ϕ are always respected. Since the rules R need not be confluent and may be highly nondeterministic, the `rewrite` and `frewrite` commands give just *one* execution path among many others. This is still very useful for execution and simulation purposes, but for analysis purposes Maude's `search` command supports a systematic breadth-first exploration of all rewrite paths until states matching a specified pattern and satisfying specified semantic conditions are reached. For example, we may want to know whether the concurrent system specified by our rewrite theory satisfies a given invariant (say, is deadlock-free). We can then search for a reachable state satisfying the *negation* of the given invariant. Within the practical limitations of time and memory, the *search* command then gives us a semi-decision procedure for the failure of such invariants, regardless of the in general infinite number of reachable states of our systems. Furthermore, for systems whose sets of reachable states are *finite*, Maude also provides a *decision procedure* for the satisfaction of linear-time temporal logic (LTL) properties. This is achieved through its built-in `MODEL-CHECKER` module which, in the experiments that we have evaluated [41,42], performs explicit-state on-the-fly model checking of LTL formulas with efficiency comparable to that of the SPIN model checker [69].

3.2 Module Algebra: The Power of Reflective Thinking

One of the most powerful features of OBJ2 and OBJ3 was the possibility of defining *parameterized modules* having semantic requirements for their instantiation specified in the form of *parameter theories*. Such modules could then be instantiated by means of *views* (theory interpretations) in the typical pushout construction way of Clear [14]. They could also be *renamed*, and instantiations and renamings could be composed in very expressive *module expressions* (see [47,65]). This supported a very powerful discipline of *parameterized programming* that inspired similar mechanisms in ML and in module interconnection languages such as LILEANNA [135]. In hindsight, however, there were two limitations. The first was that it took in practice a long time (several years of hard work) to properly implement this part of the language. Indeed, it proved to be the most complex and sophisticated component of OBJ3's LISP-based implementation. The second limitation, much less apparent to us at the time, was that OBJ's module algebra, while very powerful, was a *closed* algebra, in the sense of offering a fixed repertoire of theory operations. Of course, one could have imagined other operations, but this would have required both a new metatheory and big implementation efforts.

An important breakthrough at the theoretical level was the formulation of a general axiomatic notion of reflective logic by Manuel Clavel and myself in [23], followed by a series of papers, a Ph.D. thesis, and a book, showing that several conditional and unconditional versions of rewriting logic, as well as membership equational logic and many-sorted Horn logic with equality, are indeed reflective [24,15,16,25,26]. 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

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

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

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 $\mathcal{R}$ [24,16,83]; it is efficiently supported in the Maude implementation by means of *descent functions* [17], implemented in the built-in `META-LEVEL` module; it can be used to build a variety of theorem proving and theory transformation tools [15,20,21,27]; and it can also be used to prove metalogical properties about families of theories in rewriting logic, and about other logics represented in the rewriting logic (meta-)logical framework [5,22,6].

From the module algebra point of view, the key advantage is that the universal theory $\mathcal{U}$, and the `META-LEVEL` module that implements key descent functions for it, have a sort `Module` whose terms represent finitary rewrite theories. This means that theories become *data* that can be manipulated *within* the logic in a declarative way. Similar sorts, defining data types for parameterized modules and for views, can likewise be easily defined in extensions of the `META-LEVEL` module. In this way, Francisco Durán and I showed that many powerful theory composition operations endowing Maude with a module algebra can be defined within the logic [37,32,39]. Furthermore, the module algebra so defined now becomes easily *extensible*. For example, the notion of parameterized module, and the way in which module instantiation can be defined does not necessarily have to follow a pushout-like pattern. Different forms of parameterization, understood as new metalevel functions, can be easily defined. For instance, it is very easy to define in the Full Maude extension of Maude a `TUPLE(n)` module that for each nonzero natural number n provides a parameterized module of n-tuples [32]. Indeed, reflection has allowed considerable flexibility in easily defining and experimenting with different module composition operations before implementing some of them in the underlying Core Maude system, as has been recently done in Maude 2.2. Furthermore, Full Maude itself has been an excellent basis for building other Maude extensions such as Real-Time Maude (see Section 4.1), a strategy language for Maude [83], and the Maude termination tool (MTT) [36].

More generally, reflection has made it quite easy to build an environment of formal analysis tools for Maude. Such tools, by their very nature, manipulate and analyze rewrite theories. By reflection, a rewrite theory $\mathcal{R}$ becomes a *term* $\overline{\mathcal{R}}$ in the universal theory, which can be efficiently manipulated by the descent functions in the `META-LEVEL` module. As a consequence, Maude formal tools have a reflective design and are built in Maude as suitable extensions of the `META-LEVEL` module. They include the following:

- the Maude Church-Rosser Checker, and Knuth-Bendix and Coherence Completion tools [20,38,34,33]
- the Full Maude module composition tool [32,39]
- the Maude Predicate Abstraction tool [118]
- the Maude Inductive Theorem Prover (ITP) [16,20,27]
- the Real-Time Maude tool [109] (discussed in Section 4.1)
- the Maude Sufficient Completeness Checker (SCC) [67]
- the Maude Termination Tool (MTT) [36].

3.3 Object-Oriented Modules

A declarative treatment of the object paradigm was also a key goal from the very beginning of rewriting logic [86], and was more fully realized as part of Maude's language design in [90]. Of course, since concurrent programming was also a key goal, the point was to have a declarative way to specify and program *concurrent object systems*. This declarative approach, by using subsort overloading and proposing a key distinction between *class inheritance* and *module inheritance* solved also an old chestnut in concurrent object-oriented programming, namely the so-called *inheritance anomaly* [91].

The essential idea is extremely simple. We view the state of a concurrent object system as a "soup" of objects and messages. Mathematically, such a soup is modeled as a multiset, built up from the objects and the messages by means of a multiset union operator that is associative and commutative and has the empty multiset as its identity element. Concurrent *interactions* between objects, and between objects and messages, are then described by means of rewrite rules that transform a fragment of such a soup into a new fragment. By rewriting logic's congruence rule [88], many such rewrites can of course take place concurrently within the soup. Rules whose lefthand sides involve a single object and at most one message are called *asynchronous* and essentially correspond to the Actor model of computation [3,1]. Rules whose lefthand sides involve more than one object are called *synchronous*, because such objects have to come together synchronously in order for the interaction to take place.

More generally, the soup describing the distributed state of an object system needs not be "flat" but may instead be a "soup of soups" with arbitrary nesting depth. For example, the Internet is a network of networks and a soup of soups in exactly this sense. This structuring is very useful, for example for security and management/monitoring purposes. Carolyn Talcott and I modeled this in rewriting logic by means of our "Russian dolls" model of concurrent object reflection [100]. The "dolls" in question are meta-objects, which may contain in their belly a whole soup of other (meta-) objects, and so on "all the way down." In this way, all kinds of mechanisms for concurrent meta-object reflection can be naturally axiomatized, programmed, and reasoned about [100]. The Russian dolls model is also useful in clarifying the relationship between *object-oriented* reflection and *logical* reflection in the sense of Section 3.2. Some object-oriented reflection mechanisms do not need logical reflection: the hierarchical nesting of dolls (meta-object nesting) is enough to express them. But more powerful

concurrent object reflection mechanisms may use both the nesting of dolls and logical reflection. For example, the mobility features of Mobile Maude [35] use both meta-object reflection and logical reflection.

In Maude, concurrent object systems are specified in *object-oriented modules* [90,37,32,18]. Such modules provide syntactic sugar supporting all the usual object-oriented concepts: objects, object attributes, messages, object classes, and multiple class inheritance. Furthermore, they can be parameterized with parameter theories just like any other Maude module. Semantically, all this useful syntactic sugar can be stripped away, so that a Maude object-oriented module is semantically equivalent to an ordinary rewrite theory, that is, to a corresponding Maude *system module* into which it can be desugared. Operationally, however, knowledge of the existence of objects and messages within a multiset representing a distributed object state is used by Maude's `frewrite` command to support a rule, position, and object and message *fair* rewriting strategy. In conjunction with Maude 2.2's built-in internet sockets feature [19], this provides a very simple and elegant way of doing declarative internet programming in Maude, because there is no need whatsoever for writing any complicated thread scheduling code, which is typically needed when a conventional language is used.

4 Beyond Maude

How general and expressive is rewriting logic? The best way to find out is by pushing its limits. What follows is a progress report on how, through several research collaborations, some of us have been extending rewriting logic and its range of applications beyond those of Maude itself so as to encompass: (i) real-time and hybrid systems; (ii) probabilistic systems; (iii) deduction with logical variables; (iv) higher-order specifications; and (v) behavioral specifications.

4.1 Real-Time Maude

In many reactive and distributed systems, real-time properties are essential to their design and correctness. Therefore, the question of how systems with real-time features can be best specified, analyzed, and proved correct in the semantic framework of rewriting logic is an important one. This question has been investigated by several authors from two perspectives. On the one hand, an extension of rewriting logic called *timed rewriting logic* has been investigated, and has been applied to some examples and specification languages [71,105,125]. On the other hand, Peter Ölveczky and I have found a simple way to express real-time and hybrid system specifications *directly* in rewriting logic [106,108]. Such specifications are called *real-time rewrite theories* and have rules of the form

$$r : \{t\} \stackrel{\delta}{\longrightarrow} \{t'\} \;\; \textit{if} \;\; C$$

with δ a term denoting the *duration* of the transition (where the time domain can be chosen to be either discrete or dense), $\{t\}$ representing the *whole* state of

a system, encapsulated with $\{\_\}$, and C an equational condition. Peter Ölveczky and I have shown that, by making the clock an explicit part of the state, these theories can be *desugared* into semantically equivalent ordinary rewrite theories [106,108,109]. That is, in the desugared version we can model the state of a real-time or hybrid system as a pair $(\{t\}, \tau)$, with $\{t\}$ the current state, and with τ the current global clock time. Then the above rule becomes desugared as

$$r : (\{t\}, \tau) \longrightarrow (\{t'\}, \tau + \delta) \;\; \textit{if} \;\; C$$

Rewrite rules can then be either *instantaneous rules*, that take no time and only change some part of the state t, or *tick rules*, that advance the global time of the system according to some time expression δ and may also change the state t. When time is dense, tick rules may be *nondeterministic*, in the sense that the time δ advanced by the rule is not uniquely determined, but is instead a parametric expression (however, this time parameter is typically subjected to some equational condition C). In such cases, tick rules need a *time sampling strategy* to choose suitable values for time advance. Besides being able to show that a wide range of known real-time models (including, for example, timed automata, hybrid automata, timed Petri nets, and timed object-oriented systems) can be naturally expressed in a direct way in rewriting logic (see [108]), an important advantage of our approach is that one can use an existing implementation of rewriting logic to execute and analyze real-time specifications. Because of some technical subtleties, this seems difficult for the alternative of timed rewriting logic, although a mapping into our framework does exist [108].

Real-Time Maude [102,107,109,110] is a specification language and a formal tool built in Maude by reflection. It provides special syntax to specify real-time systems, and offers a range of formal analysis capabilities. The Real-Time Maude 2.1 tool [109,112] systematically exploits the underlying Maude efficient rewriting, search, and LTL model checking capabilities to both execute and formally analyze real-time specifications. Reflection is crucially exploited in the Real-Time Maude 2.1 implementation. On the one hand, Real-Time Maude specifications are internally desugared into ordinary Maude specifications by transforming their meta-representations. On the other, reflection is also used for execution and analysis purposes. The point is that the desired modes of execution and the formal properties to be analyzed have real-time aspects with no clear counterpart at the Maude level. To faithfully support these real-time aspects a *reflective transformational approach* is adopted: the original real-time theory and query (for either execution or analysis) are *simultaneously* transformed into a semantically equivalent pair of a Maude rewrite theory and a Maude query [109,112]. One important concern about the search and model checking analyses thus performed by Real-Time Maude is their *completeness*. Note that not all state-time pairs are visited, but only those allowed by the given time sampling strategy. For dense time it is even impossible to visit *all* times. Fortunately, under simple conditions on the specification, that are indeed satisfied by almost all examples that have been analyzed in Real-Time Maude, the analyses are indeed complete: if the tool finds no counterexamples, the given property holds [111].

In practice, Real-Time Maude executions and analyses are quite efficient. They allow scaling up to highly nontrivial specifications and case studies. In fact, both the naturalness of Real-Time Maude to specify large nontrivial real-time applications (particularly for distributed object-oriented real-time systems) and its effectiveness in simulating and analyzing the formal properties of such systems have been demonstrated in a number of substantial case studies, including: (1) the AER/NCA suite of active network protocols [102,104,113]; (2) the NORM multicast protocol [74]; (3) the OGDC wireless sensor network algorithm [134,114]; and (4) the CASH adaptive scheduling algorithm [103]. Real-Time Maude is freely available from http://www.ifi.uio.no/RealTimeMaude. It is a mature and quite efficient tool, and its source code, a tool manual, examples, case studies, and papers are all available in its web page.

4.2 PMaude and SHYMaude

Many systems are probabilistic in nature. This can be due either to the uncertainty of the environment in which they must operate, such as message losses and other failures in an unreliable environment, or to the probabilistic nature of some of their algorithms, or to both. In general, particularly for distributed systems, both probabilistic and nondeterministic aspects may coexist, in the sense that different transitions may take place nondeterministically, but the outcomes of some of those transitions may be probabilistic in nature. To specify systems of this kind, rewrite theories have been generalized to *probabilistic rewrite theories* in [72,73,4]. Rules in such theories are *probabilistic rewrite rules* of the form

$$r : t(\boldsymbol{x}) \to t'(\boldsymbol{x}, \boldsymbol{y}) \;\; \textit{if} \;\; C(\boldsymbol{x}) \;\; \textit{with probability} \;\; \boldsymbol{y} := \pi_r(\boldsymbol{x})$$

where the first thing to observe is that the term t' has new variables $\boldsymbol{y}$ disjoint from the variables $\boldsymbol{x}$ appearing in t. Therefore, such a rule is *nondeterministic*; that is, the fact that we have a matching substitution θ for the variables $\boldsymbol{x}$ such that $\theta(C)$ holds does not uniquely determine the next state fragment: there can be many different choices for the next state, depending on how we instantiate the extra variables $\boldsymbol{y}$ in t'. In fact, we can denote the different such next states by expressions of the form $t'(\theta(\boldsymbol{x}), \rho(\boldsymbol{y}))$, where θ is fixed as the given matching substitution, but ρ ranges along all the possible substitutions for the new variables $\boldsymbol{y}$. The probabilistic nature of the rule is expressed by the notation: *with probability* $\boldsymbol{y} := \pi_r(\boldsymbol{x})$, where $\pi_r(\boldsymbol{x})$ is a probability distribution *which may depend on the matching substitution* θ. We then choose the values for $\boldsymbol{y}$, that is, the substitution ρ, probabilistically according to the distribution $\pi_r(\theta(\boldsymbol{x}))$.

The fact that the probability distribution may depend on the substitution θ can be illustrated by means of a simple example. Consider a battery-operated clock. We may represent the state of the clock as a term `clock(T,C)`, with `T` a natural number denoting the time, and `C` a positive real denoting the amount of battery charge. Each time the clock ticks, the time is increased by one unit, and the battery charge slightly decreases; however, the lower the battery charge, the greater the chance that the clock will stop, going into a state of the form

`broken(T,C')`. We can model this system by means of the probabilistic rewrite rule

```
rl [tick]: clock(T,C) => if B then clock(s(T),C - (C / 1000))
                                else broken(T,C - (C / 1000))
                           fi
                  with probability B := BERNOULLI(C / 1000) .
```

that is, the probability of the clock breaking down instead of ticking normally *depends on the battery charge*, which is here represented by the battery-dependent bias of the coin in a Bernoulli trial. Note that here the new variable on the rule's righthand side is the Boolean variable `B`, corresponding to the result of tossing the biased coin. As shown in [72], probabilistic rewrite theories can express a wide range of models of probabilistic systems, including continuous-time Markov chains [131], probabilistic non-deterministic systems [119,123], and generalized semi-Markov processes [48]; they can also naturally express probabilistic object-based distributed systems [73,4], including real-time ones. Yet another class of probabilistic models that can be simulated by probabilistic rewrite theories is the class of object-based stochastic hybrid systems discussed in [99].

The PMaude language [73,4] is an experimental specification language whose modules are probabilistic rewrite theories. Note that, due to their nondeterminism, probabilistic rewrite rules *are not directly executable*. However, probabilistic systems specified in PMaude can be *simulated* in Maude. As explained in [4,93], this is accomplished by transforming a PMaude specification into a corresponding Maude specification in which actual values for the new variables appearing in the righthand side of a probabilistic rewrite rule are obtained by *sampling* the corresponding probability distribution functions using standard techniques based on random number generation and Maude's built-in `COUNTER` and `RANDOM` modules.

In general, provided that sampling for the probability distributions used in a PMaude module is supported in the underlying infrastructure, we can associate to it a corresponding Maude module. We can then use this associated Maude module to perform Monte Carlo simulations of the probabilistic systems thus specified. As explained in [4], provided all nondeterminism has been eliminated from the original PMaude module[7], we can then use the results of such Monte Carlo simulations to perform a *statistical model checking analysis* of the given system to verify certain properties. For example, for a PMaude specification of a

[7] The point is that, as explained above, in general, given a probabilistic rewrite theory and a term t describing a given state, there can be several different rewrites, perhaps with different rules, at different positions, and with different matching substitutions, that can be applied to t. Therefore, the choice of rule, position, and substitution is *nondeterministic*. To eliminate all nondeterminism, at most one rule at exactly one position and with a unique substitution should be applicable to any term t. As explained in [4], for many systems, including probabilistic real-time object-oriented systems, this can be naturally achieved, essentially by scheduling events at real-valued times that are all different, because we sample a continuous probability distribution on the real numbers.

TCP/IP protocol variant that is resistant to Denial of Service (DoS) attacks, we may wish to establish that, even if an attacker controls 90% of the network bandwidth, it is still possible for the protocol to establish a connection in less than 30 seconds with 99% probability. Properties of this kind, including properties that measure quantitative aspects of a system, can be expressed in the QATEX probabilistic temporal logic [4], and can be model checked using the VeStA tool [124]. See [2] for a substantial case study specifying a DoS-resistant TCP/IP protocol as a PMaude module, performing Monte Carlo simulations by means of its associated Maude module, and formally analyzing in VeStA its properties, expressed as QATEX specifications, according to the methodology just described. More recently, several object-based stochastic hybrid system case studies have been specified in an extension of both PMaude and Real-Time Maude called SHYMaude [99] and have been simulated in Maude. Relevant formal properties for each case study, expressed as QATEX specifications, have been statistically model checked in VeStA using Monte Carlo simulations performed in Maude [99].

4.3 Narrowing: Eqlog Revisited

Narrowing is a symbolic procedure like rewriting, except that rules, instead of being applied by matching a subterm, are applied by unifying the lefthand side with a nonvariable subterm. Traditionally, narrowing has been used as a method to solve equations in a confluent and terminating equational theory. In rewriting logic, Prasanna Thati and I have generalized narrowing to a procedure for *symbolic reachability analysis* [132]. That is, instead of solving equational goals $\exists \boldsymbol{x}.\ t = t'$, we solve reachability goals $\exists \boldsymbol{x}.\ t \longrightarrow t'$, stating that there is an instance of t from which we can reach by rewriting with rules R modulo equations E an instance of t'.

For arbitrary rewrite theories narrowing, though sound, is not a complete procedure [132]. However, for large classes of theories of interest, including theories specifying distributed object systems, narrowing is complete and provides a complete semidecision procedure for solving reachability problems [132]. Further recent work on narrowing with rewrite theories focuses on: (1) generalizing the procedure to so-called "back-and-forth narrowing," so as to ensure completeness under very general assumptions about the rewrite theory $\mathcal{R}$ [133]; and (2) efficient lazy strategies to restrict as much as possible the narrowing search space [45].

Narrowing with rewrite theories has important applications to the analysis of cryptographic protocols. A relevant point is that, since narrowing with a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is performed *modulo* the equations E, this allows more sophisticated analyses than those performed under the usual Dolev-Yao "perfect cryptography assumption." It is well-known that protocols that had been proved secure under this assumption can be broken if an attacker uses knowledge of the algebraic properties satisfied by the underlying cryptographic functions. In rewriting logic we can specify a cryptographic protocol with a type of rewrite theory $\mathcal{R} = (\Sigma, E, R)$ for which narrowing is complete, and can model those

algebraic properties as equations in E. Very recent work in this direction by Escobar, Meadows and myself [44,43] uses rewriting logic and narrowing to give a precise rewriting semantics to the inference system of one of the most effective analysis tools for cryptographic protocols, namely the NRL Protocol Analyzer [84].

Equational narrowing is a special case of rewriting logic narrowing, namely the case where we solve reachability goals of the form $\exists \boldsymbol{x}.\ equal(t, t') \longrightarrow true$ using the equations E as rewrite rules and adding the extra rule $equal(x, x) \longrightarrow true$. Furthermore, Horn logic with equality can be conservatively embedded in rewriting logic [89,81]. Indeed, in this embedding narrowing with the resulting rewrite theory is complete and agrees with SLD resolution modulo the equations E. This means that we reencounter our old friend Eqlog within the broader perspective of rewriting logic narrowing.

4.4 The Open Calculus of Constructions

Rewriting logic is an expressive *logical framework*, in which many other logics can be naturally represented [81]. Furthermore, by exploiting its reflective features in conjunction with the inductive nature of initial models, it has also good properties as a *meta-logical framework*, so that we can not only represent logics, but can also reason within the framework about their meta-logical properties [5,6].

But how good and general is it anyway? For example, how does it compare with the higher-order type theory formalisms that have been proposed by different authors as logical frameworks? Mark-Oliver Stehr and I tried to give an answer to this question using transitivity of representation mappings. If we could show that a higher-order type theory can be easily and naturally represented in rewriting logic in a conservative way, then any representation of a logic into such a type theory would automatically yield one in rewriting logic by composition. This would not be the simplest representation of that logic that one could define directly in rewriting logic, but it would prove that anything one can represent in the higher-order framework can likewise be represented in rewriting logic. Even so, some people might still be skeptical. Maybe you did it for Martin-Löf type theory, but how do I *know* that you can also do it for the Calculus of Constructions? All this could be dragged *ad nauseam*. So, what Mark-Oliver and I did in [130] was to specify a single *parametric* map (using parameterization in Maude) faithfully representing *pure type systems* (PTS) [8] into rewriting logic. Since pure type systems encompass a large class of type theories with simple types, type parameters, and type families, including the lambda cube, our skeptical colleagues would now have to come up with more exotic type theories outside the PTS general fold. At the meta-logical framework level, a careful comparison with higher-order type theories used for that purpose was given by David Basin, Manuel Clavel and myself in [6].

In fact, Mark-Oliver and I defined in [130] several representation mappings for pure type systems at different levels of abstraction. The more abstract, textbook-like representation mapped isomorphically the textbook syntax of pure type sys-

tems. But in order to give a more computational representation that would take care automatically of all the binding and substitution paraphernalia, we also gave a more concrete representation using Mark-Oliver's CINNI calculus of explicit substitutions [126] and showed it equivalent to the textbook one. Similarly, typing inference systems were represented in Maude in a computational way by means of rewrite rules [130]. This more concrete representation map was used by Mark-Oliver in his thesis [127] to implement in Maude his Open Calculus of Constructions (OCC) [127,128,129]. Since the Coquand-Huet calculus of constructions (CC) [28] is one of the instances of pure type systems, one could of course obtain an implementation of CC in Maude that way. But Mark-OIiver went considerably further. One of the sore points with higher-order type theories is their very limited and awkward way of dealing with *equalities*: an equational reasoning system like Maude can perform millions of equational deduction steps automatically in a second; but to represent such deduction steps within a given constructive type theory one needs to justify each of those equality steps constructively. By generating proof objects for the deductions of an external tool, for example for membership equational logic deduction [121], one can partly get around the problem. But Mark-Oliver's solution was more radical. By dropping the constructive interpretation and allowing simple set-theoretic models for OCC, he solved this problem directly: equality steps are allowed inside OCC, even modulo axioms like associativity and commutativity. Furthermore, OCC distinguishes several notations for equality, making clear whether they can be handled automatically by equational simplification, or need to be performed by explicit deduction steps. Likewise, a notation for relations representing rewrite rules in the rewriting logic sense is also provided. All this means that OCC can be viewed as a natural conceptual unification of the Calculus of Constructions and of rewriting logic. In particular, Maude can be naturally regarded as a sublanguage of OCC. As shown in [127,128,129], all the nice reasoning capabilities of the Calculus of Constructions, including its extensions with inductive and co-inductive principles, can be represented in OCC, that can carry out highly nontrivial proof tasks [127,128,129].

4.5 BMaude

In some sense, Maude, and languages like CafeOBJ [46] and BOBJ [54] that support hidden logic and behavioral equivalence, push the envelope in different directions of the specification language design space. Yet, there is a natural question about how these languages are all related. For example, both Maude and those languages have equational logic sublanguages. CafeOBJ itself provided some answer to this question in the form of the CafeOBJ "cube" of institutions [46], in which equational logic, hidden equational logic, and rewriting logic are related and unified. But the unification of rewriting logic and hidden logic proposed in [29] and used in [46,31] has some limitations regarding its model theory, and the matter seems to deserve further research.

While leaving open the issue of whether a more satisfactory unification of hidden logic and rewriting logic can be found, what Grigore Roşu and I did

in [96] was to develop a hidden/behavioral extension of membership equational logic called *behavioral membership equational logic*. We were interested in this extension because of theoretical and practical reasons. Theoretically, the greater generality and expressiveness of MEL over, say, order-sorted equational logic resulted in a more expressive behavioral logic. Practically, the reflective features of Maude make it easy to develop an extension of Maude called BMaude in which theories in behavioral membership equational logic can be specified as modules, and to support deduction in such modules by behavioral rewriting [120,122]. Work ahead in this direction includes passing from the present theoretical foundations and BMaude language design to a prototype implementation, and finding a more general behavioral extension of rewriting logic itself.

5 Conclusions

Science is a dialogue. This gets somewhat distorted by the unidirectional character of publications, including this one; and by the impossibility of making always explicit the many influences shaping our ideas. This festive occasion provides an opportunity for reflecting, with gratitude, on such influences; and for looking in hindsight at the road already traveled, and forward to the ways ahead. I have tried to do a little of all this from a limited and subjective perspective, but one that I am at least very familiar with: some of the ways in which the OBJ, Eqlog, and FOOPS ideas have influenced Maude. And some of the directions in which the current Maude ideas are expanding.

One way to wrap all this up is with a picture describing the relationship between the different languages I have been discussing. I call it a *language genealogy*. Solid lines describe language inclusions (or near inclusions). Dashed lines describe a weaker relationship, namely one of *influence* between different languages. Not all influences are reflected in the picture: to avoid too much cluttering, only those that I think are more *direct* are depicted. One point to bear in mind is that some of these languages are currently under construction, or even in their design phase. For example, only a first prototype of PMaude exists at present, and BMaude and SHYMaude are only language designs at this point.

Acknowledgments. In this paper I have reflected on *some* of the ways in which Joseph's ideas have influenced mine. But there are many others, both scientific and nonscientific: so much so that an actual enumeration would be both impossible and futile. It is with deep gratitude that I wish to thank Joseph, not only for his ideas and his example, but above all for his friendship. I have already mentioned by name all the colleagues who were involved in the OBJ1–3 collaborations. To all of them I also extend my sincere thanks.

Furthermore, although the references make all this clear, I want to point out that: (1) the work on Maude is joint work with all the members of the Maude team at SRI, UIUC, and the Universities of Madrid and Málaga; (2) the work on Maude tools is joint work with Manuel Clavel, Francisco Durán, Santiago Escobar, Joseph Hendrix, Salvador Lucas, Claude Marché, Hitoshi Ohsaki, Peter Ölveczky, Miguel Palomino, Ralf Sasse, and Xavier Urbain; (3) the work on real-

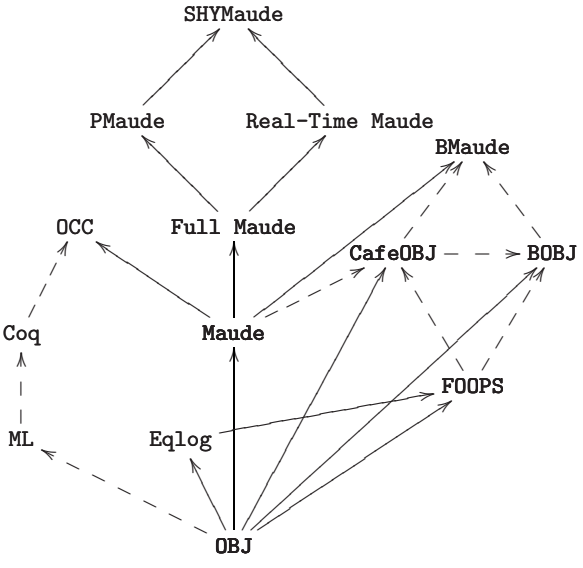

Fig. 1. A language genealogy (→ inclusion; - - > influence)

time rewrite theories is joint work with Peter Ölveczky at the University of Oslo; (4) the work on probabilistic rewrite theories and on stochastic hybrid systems is joint work with Gul Agha, Nirman Kumar, Koushik Sen, and Raman Sharykin at UIUC; (6) the work on OCC is entirely Mark-Oliver Stehr's; and (7) BMaude and its foundations are joint work with Grigore Roşu at UIUC. Several of these collaborators have also given me very useful comments to improve the final version of this paper.

This research has been supported by Grants ONR N00014-02-1-0715 and NSF CNS 05-24516, and by a bilateral CNRS-UIUC research project on "Rewriting calculi, logic and behavior."

References

1. G. Agha. *Actors.* MIT Press, 1986.
2. G. Agha, C. Gunter, M. Greenwald, S. Khanna, J. Meseguer, K. Sen, and P. Thati. Formal modeling and analysis of DoS using probabilistic rewrite theories. In *Proc. Workshop on Foundations of Computer Security (FCS'05) (Affiliated with LICS'05)*, 2005.
3. G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, 1988.
4. G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, 2005.

5. D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. In S. Kapoor and S. Prasad, editors, *FST TCS 2000*, pages 55–80. Springer LNCS, 2000.
6. D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. *ACM Transactions on Computational Logic*, 5:528–576, 2004.
7. D. Basin and G. Denker. Maude versus Haskell: an experimental comparison in security protocol analysis. In K. Futatsugi, editor, *Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications*, volume 36. ENTCS, Elsevier, 2000.
8. S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and other systems in barendregt's cube. Technical Report, Carnegie-Mellon University and Università di Torino, 1988.
9. G. Bernot, M. Bidoit, and T. Knapik. Observational specifications and the indistinguishability assumption. *Theoretical Comp. Science*, 139(1-2):275–314, 1995.
10. N. Berregeb, A. Bouhoula, and M. Rusinowitch. Observational proofs with critical contexts. In *Proceedings of FASE'98*, volume 1382 of *LNCS*. Springer, 1998.
11. M. Bidoit and R. Hennicker. Observer complete definitions are behaviourally coherent. In *OBJ/CafeOBJ/Maude at Formal Methods'99*, pages 83–94. Theta, 1999.
12. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
13. R. Bruni and J. Meseguer. Generalized rewrite theories. In J. Baeten, J. Lenstra, J. Parrow, and G. Woeginger, editors, *Proceedings of ICALP 2003, 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Springer LNCS*, pages 252–266, 2003.
14. R. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In D. Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer LNCS 86, 1980.
15. M. Clavel. Reflection in general logics and in rewriting logic, with applications to the Maude language. Ph.D. Thesis, University of Navarre, 1998.
16. M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.
17. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, Vol. 15, North Holland, 1998.
18. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
19. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.2). December 2005, http://maude.cs.uiuc.edu.
20. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000. http://maude.cs.uiuc.edu.
21. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing and J. Woodcock, editors, *FM'99 — Formal Methods*, volume 1709 of *Springer LNCS*, pages 1684–1703. Springer-Verlag, 1999.
22. M. Clavel, F. Durán, and N. Martí-Oliet. Polytypic programming in Maude. ENTCS 36, Elsevier, 2000. Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications.
23. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288, 1996. http://jerry.cs.uiuc.edu/reflection/.

24. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.
25. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285:245–288, 2002.
26. M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.
27. M. Clavel and M. Palomino. The ITP tool's manual. Universidad Complutense, Madrid, April 2005, http://maude.sip.ucm.es/itp/.
28. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
29. R. Diaconescu. Hidden sorted rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.
30. R. Diaconescu and K. Futatsugi. Behavioral coherence in object-oriented algebraic specification. *Journal of Universal Computer Science*, 6(1):74–96, 2000.
31. R. Diaconescu and K. Futatsugi. Logical foundations of CafeOBJ. *Theoretical Computer Science*, 285:289–318, 2001.
32. F. Durán. A reflective module algebra with applications to the Maude language. Ph.D. Thesis, University of Málaga, 1999.
33. F. Durán. Coherence checker and completion tools for Maude specifications. Manuscript, Computer Science Laboratory, SRI International, http://maude.cs.uiuc.edu/papers, 2000.
34. F. Durán. Termination checker and Knuth-Bendix completion tools for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, http://maude.cs.uiuc.edu/papers, 2000.
35. F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In *Agent Systems, Mobile Agents, and Applications, ASA/MA 2000*, volume 1882 of *Springer LNCS*, pages 73–85. Springer-Verlag, 2000.
36. F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving termination of membership equational programs. In P. Sestoft and N. Heintze, editors, *Proc. of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, PEPM'04*, pages 147–158. ACM Press, 2004.
37. F. Durán and J. Meseguer. An extensible module algebra for Maude. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, Vol. 15, North Holland, 1998.
38. F. Durán and J. Meseguer. A Church-Rosser checker tool for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, http://maude.cs.uiuc.edu/papers, 2000.
39. F. Durán and J. Meseguer. On parameterized theories and views in Full Maude 2.0. In K. Futatsugi, editor, *Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS 36, Elsevier, 2000.
40. S. Eker. Term rewriting with operator evaluation strategy. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, Vol. 15, North Holland, 1998.
41. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

42. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software: Proc.* 10^{th} *Intl. SPIN Workshop*, volume 2648, pages 230–234. Springer LNCS, 2003.
43. S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. Submitted for publication, 2005.
44. S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL protocol analyzer: Grammar generation. In *Proc. FMSE'05, Formal Methods in Security Engineering (Alexandria, Virginia, Nov. 2005)*, pages 1–12. ACM Press, 2005.
45. S. Escobar, J. Meseguer, and P. Thati. Natural narrowing for general term rewriting systems. In *Rewriting Techniques and Applications,* 16^{th} *Intl. Conference RTA 2005*, volume 3467, pages 279–293. Springer LNCS, 2005.
46. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
47. K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.
48. P. Glynn. The role of generalized semi-Markov processes in simulation output analysis, 1983.
49. J. Goguen. Order sorted algebra. Technical Report Semantics and Theory of Computation Report 14, UCLA, 1978.
50. J. Goguen. Types as theories. In *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991.
51. J. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In *Proceedings of WADT*, volume 785 of *LNCS*. Springer, 1994.
52. J. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics of order-sorted algebra. In W. Brauer, editor, *Proceedings, 1985 International Conference on Automata, Languages and Programming*, volume 194 of *Springer LNCS*, pages 221–231. Springer-Verlag, 1985.
53. J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting, Orsay, France, July 8-10, 1987*, pages 258–263. Springer LNCS 308, 1988.
54. J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings, 15th International Conference on Automated Software Engineering (ASE'00)*. Institute of Electrical and Electronics Engineers Computer Society, 2000. Grenoble, France, 11-15 September 2000.
55. J. Goguen and G. Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, 1999.
56. J. Goguen and G. Malcolm. A hidden agenda. *J. of TCS*, 245(1):55–101, 2000.
57. J. Goguen and J. Meseguer. Correctness of recursive flow diagram programs. In *Proc. 6th Symp. Math. Found. Comp. Sci.*, pages 580–595. Springer LNCS 53, 1977.
58. J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E. M. Schmidt, editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer LNCS 140, 1982.
59. J. Goguen and J. Meseguer. Equality, types, modules and (why not?) generics for logic programming. *Journal of Logic Programming*, 1(2):179–210, 1984.

60. J. Goguen and J. Meseguer. Models and equality for logical programming. In H. Ehrig, G. Levi, R. Kowalski, and U. Montanari, editors, *Proceedings TAPSOFT'87*, volume 250 of *Springer LNCS*, pages 1–22. Springer-Verlag, 1987.
61. J. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.
62. J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
63. J. Goguen, J. Meseguer, and D. Plaisted. Programming with parameterized abstract objects in OBJ. In D. Ferrari, M. Bolognani, and J. Goguen, editors, *Theory and Practice of Software Technology*, pages 163–193. North-Holland, 1983.
64. J. Goguen and G. Roşu. Hiding more of hidden algebra. In *Proceeding of FM'99*, volume 1709 of *LNCS*, pages 1704–1719. Springer, 1999.
65. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
66. J. A. Goguen and G. Malcolm, editors. *Software Engineering with OBJ: Algebraic Specification in Action*, volume 2 of *Advances in Formal Methods*. Kluwer Academic Publishers, Boston, 2000. ISBN 0-7923-7757-5.
67. J. Hendrix, M. Clavel, and J. Meseguer. A sufficient completeness reasoning tool for partial specifications. In *Rewriting Techniques and Applications,* 16^{th} *Intl. Conference RTA 2005*, volume 3467, pages 165–174. Springer LNCS, 2005.
68. R. Hennicker and M. Bidoit. Observational logic. In *Proceedings of AMAST'98*, volume 1548 of *LNCS*, pages 263–277. Springer, 1999.
69. G. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2003.
70. C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ3. In T. Lepistö and A. Salomaa, editors, *Proceedings, 15th Intl. Coll. on Automata, Languages and Programming, Tampere, Finland, July 11-15, 1988*, pages 287–301. Springer LNCS 317, 1988.
71. P. Kosiuczenko and M. Wirsing. Timed rewriting logic with application to object-oriented specification. Technical report, Institut für Informatik, Universität München, 1995.
72. N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical Report UIUCDCS-R-2003-2347, CS Dept., University of Illinois at Urbana-Champaign, May 2003.
73. N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model of probabilistic distributed object systems. Proc. of Formal Methods for Open Object-Based Distributed Systems, FMOODS 2003, Springer LNCS Vol. 2884, 2003.
74. E. Lien. Formal modeling and analysis of the NORM multicast protocol in Real-Time Maude. Master's thesis, Dept. of Linguistics, University of Oslo, April 2004. http://wo.uio.no/as/WebObjects/theses.woa/wo/0.3.9.
75. S. Lucas. Context-sensitive computations in functional and functional logic programs. *J. Functl. and Log. Progr.*, 1(4):446–453, 1998.
76. S. Lucas. Termination of on-demand rewriting and termination of obj programs. In *Proc. PPDP'01*, pages 82–93. ACM, 2001.
77. S. Lucas. Termination of rewriting with strategy annotations. In *Proceedings of LPAR 2001*, volume 2250 of *LNAI*, pages 669–684. Springer-Verlag, 2001.

78. S. Lucas. Context-sensitive rewriting strategies. *Inf. Comput.*, 178(1):294–343, 2002.
79. S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2005.
80. E. Manes, editor. *Proceedings of the First International Symposium on Category Theory Applied to Computation and Control, San Francisco, California, February 25–26 1974*. Springer LNCS Vol. 25, 1975.
81. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd. Edition*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Tech. Report SRI-CSL-93-05, August 1993.
82. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
83. N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proc. 5th. Intl. Workshop on Rewriting Logic and its Applications*, pages 417–441. ENTCS, Vol. 117, Elsevier, 2004.
84. C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
85. J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.
86. J. Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.
87. J. Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of the Concur'90 Conference, Amsterdam, August 1990*, pages 384–400. Springer LNCS 458, 1990.
88. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
89. J. Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Proc. 3rd Intl. Conf. on Algebraic and Logic Programming*, pages 158–200. Springer LNCS 632, 1992.
90. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
91. J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. M. Nierstrasz, editor, *Proc. ECOOP'93*, pages 220–246. Springer LNCS 707, 1993.
92. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.
93. J. Meseguer. A rewriting logic sampler. In *Proc. International Colloquium on Theoretical Aspects of Computing ICTAC05 (Hanoi, Vietnam, October 2005)*, volume 3722 of *LNCS*, pages 1–28. Springer, 2005.
94. J. Meseguer and J. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.
95. J. Meseguer and G. Roşu. A total approach to partial algebraic specification. In *Proc. ICALP'02*, pages 572–584. Springer LNCS 2380, 2002.
96. J. Meseguer and G. Roşu. Towards behavioral Maude: Behavioral membership equational logic. In *Proc. CMCS'02*. Elsevier ENTCS, 2002.

97. J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In *Proc. Intl. Joint Conf. on Automated Reasoning IJCAR'04, Cork, Ireland, July 2004*, pages 1–44. Springer LNAI 3097, 2004.
98. J. Meseguer and G. Roşu. The rewriting logic semantics project. In *Proc. SOS 2005*. Elsevier ENTCS, 2005.
99. J. Meseguer and R. Sharykin. Specification and analysis of distributed object-based stochastic hybrid systems. In *Hybrid Systems, HSCC 2006*, pages 460–475. Springer LNCS 3927, 2006.
100. J. Meseguer and C. Talcott. Semantic models for distributed object reflection. In *Proceedings of ECOOP'02, Málaga, Spain, June 2002*, pages 1–36. Springer LNCS 2374, 2002.
101. J. Moore, R. Krug, H. Liu, and G. Porter. Formal models of Java at the JVM level – a survey from the ACL2 perspective. In *Proc. Workshop on Formal Techniques for Java Programs, in association with ECOOP 2001*, 2002.
102. P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000. http://maude.csl.sri.com/papers.
103. P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In *Proc.* FASE 2006, LNCS 3922, pages 357–372. Springer, 2005.
104. P. C. Ölveczky, M. Keaton, J. Meseguer, C. Talcott, and S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In *Proc. of FASE'01, 4th Intl. Conf. on Fundamental Approaches to Software Engineering*, volume 2029 of *Springer LNCS*, pages 333–348. Springer-Verlag, 2001.
105. P. C. Ölveczky, P. Kosiuczenko, and M. Wirsing. An object-oriented algebraic steam-boiler control specification. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *The Steam-Boiler Case Study Book*, pages 379–402. Springer-Verlag, 1996. Vol. 1165.
106. P. C. Ölveczky and J. Meseguer. Specifying real-time systems in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.
107. P. C. Ölveczky and J. Meseguer. Real-Time Maude: a tool for simulating and analyzing real-time and hybrid systems. volume 36. ENTCS, Elsevier, 2000. Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications.
108. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.
109. P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. In N. Martí-Oliet, editor, *Proc. 5th. Intl. Workshop on Rewriting Logic and its Applications*, pages 285–314. ENTCS, Vol. 117, Elsevier, 2004.
110. P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In T. Margaria and M. Wermelinger, editors, *Fundamental Approaches to Software Engineering (FASE 2004)*, volume 2984 of *Springer LNCS*, pages 354–358. Springer-Verlag, 2004.
111. P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. In G. Denker and C. Talcott, editors, *Proc. 6th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2006.
112. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 2006. To appear.

113. P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Technical Report UIUCDCS-R-2004-2467, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004. Available at http://www.ifi.uio.no/RealTimeMaude.
114. P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In *The 14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS) 2006*. IEEE Computer Society Press, 2006.
115. P. Padawitz. Swinging data types: Syntax, semantics, and theory. In *Proceedings, WADT'95*, volume 1130 of *LNCS*, pages 409–435. Springer, 1996.
116. P. Padawitz. Towards the one-tiered design of data types and transition systems. In *Proceedings of WADT'97*, volume 1376 of *LNCS*, pages 365–380. Springer, 1998.
117. P. Padawitz. Swinging types = functions + relations + transition systems. *Theoretical Computer Science*, 243:93–165, 2000.
118. M. Palomino. A predicate abstraction tool for Maude. Manuscript, Universidad Complutense, 2005, http://maude.sip.ucm.es/~miguelpt/papers/pa-tool.pdf.
119. M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.
120. G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
121. G. Roşu, S. Eker, P. Lincoln, and J. Meseguer. Certifying and synthesizing membership equational proofs. In *Proc. FM'03*, volume 2805, pages 359–380. Springer LNCS, 2003.
122. G. Roşu and J. Goguen. Hidden congruent deduction. In *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *LNAI*. Springer, 2000.
123. R. Segala. *Modelling and Verification of Randomized Distributed Real Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.
124. K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *17th conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 266–280, Edinburgh, Scotland, 2005. Springer.
125. L. Steggles and P. Kosiuczenko. A timed rewriting logic semantics for SDL: a case study of the alternating bit protocol. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, ENTCS, Vol. 15, North Holland, 1998.
126. M.-O. Stehr. CINNI - a generic calculus of explicit substitutions and its application to lambda-, sigma- and pi-calculi. ENTCS 36, Elsevier, 2000. Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications.
127. M.-O. Stehr. Programming, Specification, and Interactive Theorem Proving — Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory. Doctoral Thesis, Universität Hamburg, Fachbereich Informatik, Germany, 2002. http://www.sub.uni-hamburg.de/disse/810/.
128. M.-O. Stehr. The Open Calculus of Constructions: An equational type theory with dependent types for programming, specification, and interactive theorem proving (Part I). *Fundamenta Informaticae*, 68(1–2):131–174, 2005.
129. M.-O. Stehr. The Open Calculus of Constructions: An equational type theory with dependent types for programming, specification, and interactive theorem proving (Part II). *Fundamenta Informaticae*, 68(3):249–288, 2005.
130. M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic: Specifying typed higher-order languages in a first-order logical framework. In *Essays in Memory of Ole-Johan Dahl*, pages 334–375. Springer LNCS Vol. 2635, 2004.

131. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton, 1994.
132. P. Thati and J. Meseguer. Symbolic reachability analysis using narrowing and its application to the verification of cryptographic protocols. In N. Martí-Oliet, editor, *Proc. 5th. Intl. Workshop on Rewriting Logic and its Applications*, pages 153–182. ENTCS, Vol. 117, Elsevier, 2004.
133. P. Thati and J. Meseguer. Complete symbolic reachability analysis using back-and-forth narrowing. In *Proceedings of CALCO 2005*, volume 3629 of *LNCS*, pages 379–394. Springer, 2005.
134. S. Thorvaldsen and P. C. Ölveczky. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. http://www.ifi.uio.no/RealTimeMaude/OGDC, 2005.
135. W. Tracz. Parametrized programming in LILEANNA. In *Proc. 1993 ACM/SIGAPP Symp. on Applied Computing (SAC '93)*, pages 77–86, 1993.
136. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.