

# Executable Computational Logics: Combining Formal Methods and Programming Language Based System Design\*

José Meseguer  
Computer Science Department  
University of Illinois at Urbana-Champaign  
Urbana, IL 61801, USA  
meseguer@cs.uiuc.edu

## Abstract

*An executable computational logic can provide the desired bridge between formal system properties and formal methods to verify them on the one hand, and executable models of system designs based on programming languages on the other. However, not all such logics are equally well suited for the task. This paper gives some requirements that seem important for a computational logic to be suitable in practice, and discusses the experience with rewriting logic, its Maude language implementation, and its formal tool environment, concluding that they seem to meet well those requirements.*

## 1. The General Idea

The present conference explores a convergence of formal methods and programming language based approaches to system design in both hardware and embedded hardware-software systems. There are many practical benefits to be gained from such a convergence and, furthermore, there are important research issues involved.

I wish to put forward a simple general idea that in my view provides a conceptual key to exploring a convergence of this kind. As suggested in the title, the idea is that an *executable computational logic* can provide the desired bridge between formal system properties and formal methods to verify them on the one hand, and executable models of system designs based on programming languages on the other.

The *general* idea as such is hardly new: it has already been demonstrated very successfully by different researchers using different computational logics. There are

---

\*Research supported by ONR Grant N00014-02-1-0715, NSF Grant CCR-0234524, and by DARPA through Air Force Research Laboratory Contract F30602-02-C-0130.

however specific *requirements* that the tasks of designing and verifying hardware and embedded hardware-software systems and, as the list of topics in this conference suggests, also distributed systems involving subsystems of this kind pose upon an executable computational logic. Such requirements can make some logics *more suitable* than others in certain respects.

To explore this proposal in more detail, a first useful distinction is one 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 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 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 non-executable 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.

The above distinction brings us also to the heart of the problem addressed by this conference: how can we *formally*, that is using logical and mathematical methods, verify a system 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. Mathematics requires a precise and unambiguous notation, and *mechanizable* mathematics supported by for-

mal tools make this requirement absolutely necessary. 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.

But we should not despair. What may be a *de facto* undesirable situation is not by any means a logical necessity. Indeed, here is where executable computational logics can render an invaluable service. They can either:

1. be used as a *declarative programming language* with a *precise mathematical semantics* to express system specifications; or
2. be used to *give a precise mathematical semantics* to a conventional programming language.

In the first case, the gap between formal property specifications and informal system specifications evaporates: we now have a *formal* system specification as well. In the second case, the gap is bridged by the executable computational logic, which associates the desired mathematical model to what hitherto was only an informal specification. The first option points to a more revolutionary path, whereas the second shows a sound evolutionary way out of the quandary.

I have not yet defined what I mean by an *executable computational logic* and some clarification is perhaps needed. 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*. 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; and Maude is a declarative programming language associated to rewriting logic.

One can always blur the above distinctions, but this is no very helpful. For example, there is always the Quixotic and amusing possibility of declaring that *everything is a logic!* including, say, C<sup>++</sup>, arriving at a toothless notion of “logic”. The possibilities for confusion and obscurantism are indeed endless; but such verbal games are for the most part a waste of time and I will not waste mine or the reader’s playing games in this paper. For a general *mathematical* notion of logic and general axiomatic requirements for declarative programming languages see [39].

## 1.1. Some Requirements

Logics and programming languages are both means to an end. The end is *representation* of some desired entity, for example a system design. Specific representational goals make some logics or languages *more suitable* than others for the goals in question. In our case, an interesting question is: what *representational capabilities* of a computational logic would be particularly desirable to express designs of hardware systems, embedded hardware-software systems, and distributed systems made out of them?

Without in any way trying to be exhaustive—indeed, I welcome suggestions on other requirements as well as criticism of the following ones—I would like to list some representational capabilities of a computational logic that I think are both relevant and important:

1. good *data representation* capabilities,
2. support for *concurrency and nondeterminism*,
3. support for *real-time* and *hybrid* computations,
4. capable of representing *probabilistic* behavior,
5. *simplicity* of the formalism
6. *efficient* implementability,
7. *initial model* semantics.

My justification for the above requirements is as follows. Without good data representation capabilities there will be a serious gap of understandability; for example representing numbers as Church numerals is not acceptable. Concurrency and nondeterminism are pervasive even for hardware implementing a *sequential* microprocessor, because of performance gains through parallelization; they are essential for distributed systems. Support for real time and hybrid computation is essential for embedded systems. Representation of probabilistic behavior is important to model faults and measure performance, and is key for networked embedded systems, wireless communication, and many other applications. Simplicity is a key feature for a formalism to be widely adopted: system engineers are allergic to baroque formalisms. Without an *efficient* implementation as a programming language a computational logic will not be able to compete with conventional programming languages and will be unusable for large designs. Last, but not least, if a theory  $T$  in the logic has an *initial model*, say  $\mathcal{I}_T$ , then this provides the *intended mathematical model* for the system design specified by  $T$ . Then, to say that the design satisfies a *property*<sup>1</sup>  $\varphi$  means precisely that the satisfaction relation  $\mathcal{I}_T \models \varphi$  holds.

<sup>1</sup>Note that the logic in which we express the property  $\varphi$  need not be the computational logic of the theory  $T$ . Indeed, we have already pointed

## 1.2. Some Previous Approaches

As already pointed out, the idea of using an executable computational logic to specify hardware and hardware-software systems is hardly new. Indeed, this idea has already been shown to be highly effective. This paper is not a survey and I will not try to give one. I will, instead, illustrate with examples how a family of computational logics, namely both first-order and higher-order *equational logics* have indeed been used very successfully for hardware and hardware-software system specification. Even in this sub-area no attempt at covering all the relevant work is made: I just give a few illustrative examples:

- Using the ACL2 logic, an executable equational logic based on a purely functional subset of Common Lisp, and the ACL2 inductive theorem prover [33], researchers in the ACL2 community have specified an impressive collection of hardware systems, including industrial designs (for some case studies see [32]). In fact, ACL2 is routinely used in industry for hardware specification and verification.
- Using the higher-order equational logic of the lazy typed lambda calculus and its implementation in Haskell [23], researchers in the Hawk project at OGI have specified and reasoned about superscalar micro-processor designs [38].
- Using the OBJ language [26], based on order-sorted equational logic, several researchers have specified and verified various hardware designs [25, 50, 20, 22].

What all these equational computational logics and languages have in common is the *Church-Rosser assumption*, which guarantees a *unique final result* if equational simplification terminates. That is, these logics are ideally suited to specify *deterministic systems* and have been shown to be very successful in dealing with deterministic hardware systems of this kind. However, in the context of the broader requirements discussed in Section 1.1, equational logics have serious limitations. For example, concurrency and nondeterminism cannot be directly supported. Different tricks to eliminate nondeterminism can be used to get around this problem: introducing a scheduler, considering the set of all behaviors as a computational object, etc. But these tricks are unsatisfactory and have serious limitations, particularly for distributed systems.

---

out that, in general, properties may be specified in logics that are not executable in our sense. The property  $\varphi$  may belong to a superlogic of the computational logic or, more generally, to a logic related to the computational logic by a suitable map of logics in the sense of [39].

## 2. The Rewriting Logic/Maude Experience

Rewriting logic [40] is a computational logic that fits quite well the requirements discussed in Section 1.1. Indeed, one of its key features is its natural and general support for concurrency and nondeterminism, since it can naturally express a very wide range of concurrency models [40, 42, 37]. A concurrent system is axiomatized by a *rewrite theory*  $\mathcal{R} = (\Sigma, E, R)$ , where  $(\Sigma, E)$  is an equational theory describing its set of *states* as the algebraic data type  $T_{\Sigma/E,k}$  associated to the initial algebra  $T_{\Sigma/E}$  of  $(\Sigma, E)$  by the choice of a type  $k$  of states in  $\Sigma^2$ . The system's *transitions* are axiomatized by the *conditional rewrite rules*  $R$  which are of the form,

$$l : t \longrightarrow t' \text{ if } \text{cond}$$

with  $l$  a label,  $t$  and  $t'$   $\Sigma$ -terms, possibly with variables, and *cond* a condition. Intuitively, a rule of this form specifies a pattern  $t$  so that, if some fragment of the system's state matches that pattern and satisfies the condition *cond*, then a local transition of that state fragment changing into the pattern  $t'$  can take place. Under reasonable assumptions about  $E$  and  $R$ , rewrite theories are *executable*. Indeed, there are several rewriting logic language implementations, including ELAN [7], CafeOBJ [24], and Maude [12]. Both ELAN and Maude have high-performance implementations achieving millions of rewrites per second on realistic applications.

We will assume that the equations of a rewrite theory are decomposed as a union  $E \cup A$ . We then say that the rewrite theory  $\mathcal{R} = (\Sigma, E \cup A, R)$  is *executable* if: (1) there exists a *matching algorithm modulo* the equational axioms  $A$ <sup>3</sup>; (2) the equational theory  $(\Sigma, E \cup A)$  is (ground) *Church-Rosser and terminating modulo*  $A$  [15]; and (3) the rules  $R$  are (ground) *coherent* [53] relative to the equations  $E$  modulo  $A$ . The coherence requirement ensures that equations and rules work well together: in particular a rewrite can always be performed by first obtaining the canonical form of a term  $t$  by simplifying it with the equations  $E$  modulo  $A$  and then rewriting that canonical form with a rule in  $R$ .

One obvious but important observation is that *rewriting logic includes equational logic as a sublogic*. Indeed, if the set  $R$  of rewrite rules is empty, an executable rewrite theory  $\mathcal{R}$  becomes an equational theory  $(\Sigma, E \cup A)$  which is (ground) Church-Rosser and terminating modulo  $A$  and

---

<sup>2</sup>We can allow very general equational theories in *membership equational logic* [41], that can have types, subtypes defined by semantic conditions, and operator overloading. The desired set of states is then described by the carrier  $T_{\Sigma/E,k}$  of the initial algebra  $T_{\Sigma/E}$  for one of those types  $k$ , technically called either *sorts* or *kinds* in [41]. The elements of  $T_{\Sigma/E}$  are  $E$ -equivalence classes of terms  $[t]_E$ ; that is, two terms are equal iff they can be proved so by  $E$ .

<sup>3</sup>In Maude, the axioms  $A$  for which the rewrite engine supports matching modulo are any combination of *associativity*, *commutativity*, and *identity* axioms for different operators.

can be efficiently executed as a program. In Maude, this equational sublogic gives rise to the sublanguage of *functional modules*. This means that all the good properties of equational logic as a computational logic for specifying deterministic systems discussed in Section 1.2 are automatically inherited by rewriting logic. In particular, this yields very good data representation capabilities, since the logic supports arbitrary algebraically specified data types *modulo* axioms  $A$  such as associativity, commutativity, and identity. Indeed, Maude supports rewriting modulo such axioms very efficiently, making such rewriting a serious possibility for real programming [21].

Rewriting logic is a simple, easy to understand logic. Its equational sublogic is very simple: the usual algebraic replacement of equals for equals. And a rewrite rule  $l : t \longrightarrow t' \text{ if } \textit{cond}$  axiomatizes the simple notion of a *local transition* guarded by  $\textit{cond}$  and parameterized by the variables in  $t, t', \textit{cond}$ . For example, in our experience network engineers seem to understand formal specifications of a network protocol in Maude more easily than corresponding informal use case specifications. Finally, a rewrite theory  $\mathcal{R}$  always has an *initial model*  $\mathcal{I}_{\mathcal{R}}$  [40, 9], so that satisfaction of properties means exactly satisfaction in the intended model  $\mathcal{I}_{\mathcal{R}}$ . This completes the discussion of how requirements 1–2 and 5–7 in Section 1.1 are met by rewriting logic and its Maude implementation.

## 2.1. Real-Time and Probabilistic Rewrite Theories

Requirements 3–4 are respectively met by *real-time* rewrite theories and *probabilistic* rewrite theories. In a real-time rewrite theory [48] rewrite rules have the form,

$$l : t \longrightarrow t' \text{ if } \textit{cond} \text{ in time } \tau$$

where  $\tau$  is a term of type *Time*.  $\tau$  may contain some among the variables  $\vec{x}$  in  $t, t', \textit{cond}$ . Intuitively,  $\tau$  indicates the *duration* or *time advance* of the rewrite, which is not fixed, but may depend on the substitution  $\theta$  for the variables  $\vec{x}$  in each rewrite. *Time* is an algebraic data type of *time values*, which can be either discrete or continuous depending on the applications. Two important facts about real-time rewrite theories are:

1. They are *reducible to ordinary rewrite theories* [48], so that the above notation can be viewed as syntactic sugar for a special class of rewrite theories containing a *Time* data type and satisfying some additional requirements.
2. Many different computational models of real-time and hybrid systems such as, for example, timed automata [3], hybrid automata [2], and timed Petri nets [51] can be naturally specified by corresponding classes of real-time rewrite theories [48].

Rules in a *probabilistic rewrite theory* [34] are of the form,

$$l : t(\vec{x}) \longrightarrow t'(\vec{x}, \vec{y}) \text{ if } \textit{cond}(\vec{x}) \text{ with probability } \pi(\vec{x})$$

By making explicit the different variables  $\vec{x}$  and  $\vec{y}$  appearing in the terms  $t, t'$  and in the condition  $\textit{cond}$  we can see that the rule itself is *nondeterministic*. The rule will match a state fragment if there is a substitution  $\theta$  for the variables  $\vec{x}$  making  $\theta(t)$  equal to that state fragment, and, furthermore, such that  $\theta(\textit{cond})$  is satisfied. But since  $t'$  is of the form  $t'(\vec{x}, \vec{y})$ , the next state is not uniquely determined: it depends on the choice of an additional substitution  $\rho$  for the variables  $\vec{y}$ . This choice of  $\rho$  is then made according to the probability distribution  $\theta(\pi)$ , which in general is not a fixed function, but a *family* of functions: one for each match  $\theta$  of the variables  $\vec{x}$ .

It turns out that this general notion of probabilistic rewrite theory can be used to specify in a natural way many different models of probabilistic systems such as, for example, probabilistic nondeterministic systems [6, 14], probabilistic Petri nets [35, 36], and probabilistic algebra approaches [31, 30].

Probabilistic rewrite theories support the system specification level. At the property specification level, [34] proposes a *probabilistic rewriting temporal logic* (PRTL and PRTL\*) as a unifying temporal logic. PRTL and PRTL\* are probabilistic extensions of CTL and CTL\* [11], respectively, and are designed to express properties of probabilistic rewrite theories. A number of probabilistic temporal logics proposed for different models, including pCTL and pCTL\* [6], PBTL [5], and CSL [1, 4, 29] can be represented as special cases of PRTL and PRTL\*.

## 2.2. Properties: Maude’s Formal Tool Environment

The fact that, under reasonable assumptions, rewriting logic specifications are executable allows us to have a flexible range of *increasingly stronger formal methods* to formally analyze and verify properties of a system. Only after less costly and “lighter” methods have been used, it is meaningful and worthwhile to invest effort on “heavier” and costlier methods. A rewriting logic language implementation, together with an associated environment of formal tools, can be used to support, among others, the following, increasingly stronger methods: (1) formal specification, (2) execution of the specification, (3) search-based analysis of the state space, (4) model-checking analysis, and (5) formal proof.

Maude and its tool environment support the above methods (1)–(5). Methods (1)–(4) are supported by Maude itself: (1)–(2) through module input and its execution commands; (3) through its `search` command, which allows searching for a reachable state satisfying a given pattern

and condition and can be used to find counterexamples to safety properties even in infinite-state systems; (4) through its built-in linear time temporal logic (LTL) model checker [19]; and (5) through its inductive theorem prover (ITP) and tools to check the Church-Rosser property, coherence, and termination, and to perform Knuth-Bendix completion [13, 18, 16, 17]; the ITP and the above tools can also be used in conjunction with the LTL model checker to prove LTL properties of an infinite-state system by defining a finite-state *abstraction* of the system, model checking the abstraction, and discharging the abstraction correctness proof obligations with the proving tools [44]. Furthermore, for real-time rewrite theories, methods (1)–(4) are also supported in the Real-Time Maude tool [47, 46]; and for probabilistic rewrite theories methods (1)–(2) are currently supported by the PMaude prototype [34]. Finally, Grigore Roşu and his collaborators have developed several “lightweight” formal methods using Maude and the Maude ITP such as: runtime program verification and monitoring [28], domain-specific program certification [10], and program-with-proof synthesis [49]. These are very practical and scalable “pushbutton” methods that can formally analyze and verify large programs.

Of course, the *properties* analyzed by these different methods can be expressed in logics different from rewriting logic which need not be executable. For example, in method (4) they can be properties in linear time temporal logic [19], and some theorem proving tasks involve proving properties in first-order logic enriched with inductive principles [13]. The point, however, is that the *intended model* satisfying the given properties is either the initial model  $\mathcal{I}_{\mathcal{R}}$  of the system specification  $\mathcal{R}$ , or a closely related model (for example a Kripke structure) associated to  $\mathcal{I}_{\mathcal{R}}$  in an appropriate map of logics.

### 3. Conclusions

The main idea put forward in this paper is that an *executable computational logic* can provide the desired bridge between formal system properties and formal methods to verify them on the one hand, and executable models of system designs based on programming languages on the other. This idea is not new at all, but the current state of affairs in system design and the existing gap between formal methods and designs based on conventional programming languages make it worth repeating.

However, the question is not just having *some* computational logic, but rather carefully selecting logics that are well suited for the tasks at hand. I have proposed some requirements that seem important in practice, and have discussed how equational logics, while very effective for specifying in an executable way deterministic systems, do not seem to meet other requirements. I have also explained why

rewriting logic extends equational logic in a way that makes it a suitable computational logic meeting requirements 1–7 rather well. In particular, the Maude implementation and its Real-Time Maude and PMaude extensions, as well as the Maude formal tool environment support not only efficient simulation of system designs, but also a wide spectrum of formal methods to verify system properties.

A point touched upon in Section 1 but worth discussing further is the evolutionary possibility of bridging the gap between formal methods and conventional language based system designs by using a declarative language like Maude to give a formal semantics to a conventional language of interest and then reasoning about the given design within Maude. This is a real possibility; in ACL2 a similar approach has recently been taken quite successfully to reason about Java programs [45]. Also, thanks to the recent work of Braga [8], Verdejo [52], and Roşu et al. [10], there is by now ample evidence that it is quite easy to give an executable formal semantics to any programming language in Maude with reasonable efficiency.

My main point is of course quite general, and applies to other computational logics that could be proposed. However, rewriting logic and Maude seem promising as one possible approach. As always, more experience is needed to fully ascertain this in practice. However, the current experience, including applications to pipelined microprocessor specification and verification [27], real-time specifications and scheduling analyses [46], and many network protocol applications (see references in [43, 37]) gives some grounds for optimism.

### Acknowledgments

As the references make clear, the work on rewriting logic and Maude is a *collective* effort. I am very grateful to all my colleagues in the Maude team for their work on Maude, its theoretical foundations, and its tools. The ideas on real-time rewrite theories are joint work with Peter Ölveczky; those on probabilistic rewrite theories are joint work with Nirman Kumar, Koushik Sen, and Gul Agha. More generally, I am very grateful to all the researchers who have advanced the rewriting logic research program and whose work (up to 2001) is summarized in the bibliography of [37]. Grigore Roşu, besides having made important contributions to this research program, deserves special thanks for carefully reading a draft of this paper and giving helpful suggestions. I am also very grateful to the organizers of MEMOCODE’03 for giving me the opportunity of presenting these ideas in such an excellent conference and at such a magic place.

## References

- [1] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time Markov chains. In Rajeev Alur and Thomas A. Henzinger, editors, *Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 269–276, New Brunswick, NJ, USA, 1996. Springer Verlag.
- [2] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R. Grossman, A. Nerode, A. Ravn, and H. Rischel, editors, *Workshop on Theory of Hybrid Systems*, pages 209–229. Springer LNCS 739, 1993.
- [3] R. Alur and D. Dill. The theory of timed automata. In J. de Bakker, G. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, 1991.
- [4] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous-time Markov chains. *ACM Transactions on Computational Logic (TOCL)*, 1(1):162–170, 2000.
- [5] C. Baier and M. Z. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.
- [6] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 15, 1995.
- [7] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.
- [8] C. Braga. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.
- [9] R. Bruni and J. Meseguer. Generalized rewrite theories. To appear in Proc. ICALP’03, Springer LNCS, <http://maude.cs.uiuc.edu>.
- [10] F. Chen, G. Rosu, and R. P. Venkatesan. Rule-based analysis of dimensional safety. In *Rewriting Techniques and Applications (RTA’03)*, Lecture Notes in Computer Science, to appear. Springer-Verlag, 2003.
- [11] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.
- [12] 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.
- [13] 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>.
- [14] L. de Alfaro. Temporal logics for the specification of performance and reliability. In *Symposium on Theoretical Aspects of Computer Science*, pages 165–176, 1997.
- [15] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. North-Holland, 1990.
- [16] F. Durán. Coherence checker and completion tools for Maude specifications. Manuscript, Computer Science Laboratory, SRI International, <http://maude.cs.uiuc.edu/papers>, 2000.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] S. M. Eker. *Formal Foundations for the Design of Rasterization Algorithms and Architectures*. PhD thesis, School of Computer Studies, University of Leeds, 1990.
- [21] S. M. Eker. Associative-commutative rewriting on large terms. In *Rewriting Techniques and Applications (RTA’03)*, Lecture Notes in Computer Science, to appear. Springer-Verlag, 2003.
- [22] S. M. Eker, V. Stavridou, and J. V. Tucker. Verification of synchronous concurrent algorithms using OBJ3: A case study of the Pixel-Planes architecture. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, pages 231–252. Springer-Verlag, 1991.
- [23] P. H. et al. Report on the programming language Haskell. Technical report, Computing Science Department, University of Glasgow, April 1990.
- [24] K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
- [25] J. Goguen. OBJ as a theorem prover with application to hardware verification. In P. Subramanyam and G. Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267. Springer-Verlag, 1989.
- [26] 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.
- [27] N. Harman. Verifying a simple pipelined microprocessor using Maude. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques*, volume 2267 of *Lecture Notes in Computer Science*, pages 128–151. Springer-Verlag, 2001.
- [28] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE’01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
- [29] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov chain model checker. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 347–362, 2000.
- [30] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

- [31] J. Hillston and M. Ribaud. Stochastic process algebras: A new approach to performance modeling. In J. W. K. Bagchi and G. Zobrist, editors, *Modeling and Simulation of Advanced Computer Systems*. Gordon Breach, 1998.
- [32] M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer, 2000.
- [33] M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.
- [34] N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. manuscript, March 2003. CS Dept., University of Illinois at Urbana-Champaign.
- [35] M. A. Marsan. Stochastic Petri nets: An elementary introduction. *Lecture Notes in Computer Science; Advances in Petri Nets 1989*, 424:1–29, 1990. NewsletterInfo: 36.
- [36] M. A. Marsan, A. Bobbio, and S. Donatelli. Petri nets in performance analysis: An introduction. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:211–256, 1998.
- [37] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
- [38] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Proceedings of the International Conference on Computer Languages*. IEEE, 1988.
- [39] J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium '87*, pages 275–329. North-Holland, 1989.
- [40] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
- [41] 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.
- [42] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*. Springer-Verlag, 1999.
- [43] J. Meseguer. Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems, FMOODS 2000*, pages 89–117. Kluwer, 2000.
- [44] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. To appear in Proc. CADE 2003, Springer LNCS, <http://maude.cs.uiuc.edu>.
- [45] 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.
- [46] 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>.
- [47] P. C. Ölveczky and J. Meseguer. Real-Time Maude: a tool for simulating and analyzing real-time and hybrid systems. ENTCS, Elsevier, 2000. Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications.
- [48] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.
- [49] G. Rosu, R. P. Venkatesan, J. Whittle, and L. Leustean. Certifying optimality of state estimation programs. In *Computer Aided Verification (CAV'03)*. Springer, 2003. to appear in Lecture Notes in Computer Science.
- [50] V. Stavridou. Specifying in OBJ, verifying in REVE and some ideas about time. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 171–191. Kluwer, 2000.
- [51] W. van der Aalst. Interval timed coloured Petri nets and their analysis. In M. A. Marsan, editor, *Application and Theory of Petri Nets 1993*, pages 453–472. Springer LNCS 691, 1993.
- [52] A. Verdejo. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Departamento de Informática, Pontificia Universidade Católica de Rio de Janeiro, Brasil, 2001.
- [53] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.