The Rewriting Logic Semantics Project: A Progress Report

José Meseguer[a], Grigore Roşu[b]

[a]*Department of Computer Science,*
University of Illinois at Urbana-Champaign, USA
meseguer@illinois.edu
[b]*Department of Computer Science,*
University of Illinois at Urbana-Champaign, USA, and
University Alexandru Ioan Cuza, Iaşi, Romania
grosu@illinois.edu

Abstract

Rewriting logic is an executable logical framework well suited for the semantic definition of languages. Any such framework has to be judged by its effectiveness to bridge the existing gap between language definitions on the one hand, and language implementations and language analysis tools on the other. We give a progress report on how researchers in the rewriting logic semantics project are narrowing the gap between theory and practice in areas such as: modular semantic definitions of languages; scalability to real languages; support for real time; semantics of software and hardware modeling languages; and semantics-based analysis tools such as static analyzers, model checkers, and program provers.

Keywords: Rewriting Logic, Programming Languages, Semantics, Maude, K

1. Introduction

The disconnect between theory and practice is one of the worse evils in computer science. Theory disconnected from practice becomes irrelevant; and practice without theory becomes brute-force, costly and ad-hoc engineering. One of the current challenges in formal approaches to language semantics is precisely how to effectively bridge the gap between theory and practice. There are two distinct dimensions to this gap:

(1) Given a language $\mathcal{L}$, there is often a substantial gap between: (i) a formal semantics for $\mathcal{L}$; (ii) an implementation of $\mathcal{L}$; and (iii) analysis tools for $\mathcal{L}$, including static, dynamic, and deductive tools.
(2) Even if a formal semantics exists for a programming language $\mathcal{L}$, there may not be any formal semantics available at the higher level of software designs and models, or at the lower level of hardware.

Regarding (1), a semantics of $\mathcal{L}$ may just be a "paper semantics," such as some SOS rules on a piece of paper; or it may be a "toy semantics," not for $\mathcal{L}$

itself, but for a greatly simplified sub-language. Furthermore, the way a compiler for $\mathcal{L}$ is written may have no connection whatever with a formal semantics for $\mathcal{L}$, so that different compilers provide different language behaviors. To make things worse, program analysis tools for $\mathcal{L}$, including tools that supposedly provide some formal analysis, may not be systematically based on a formal semantics either, so that the confidence one can place of the answers from such tools is greatly diminished. Regarding (2), one big problem is that software modeling notations often lack a formal semantics. A related problem is that this lack of semantics manifests itself as a lack of *analytic power*, that is, as an incapacity to uncover expensive design errors which could have been caught by formal analysis.

We, together with many other colleagues all over the world, have been working for years on the *rewriting logic semantics project* (see [1, 2, 3] for some overview papers at different stages of the project). The goal of this project is to substantially narrow the gap between theory and practice in language specifications, implementations and tools, in both of the above dimensions (1)–(2). In this sense, rewriting logic semantics is a *wide-spectrum framework*, where:

1. The formal semantics of a language $\mathcal{L}$ is used as the *basis* on which both language implementations and language analysis tools are built.
2. The same semantics-based approach is used not just for programming languages, but also for software and hardware modeling languages.

Any attempt to bridge theory and practice cannot be judged by theoretical considerations alone. One has to evaluate the practical effectiveness of the approach in answering questions such as the following:

- *Executability.* Is the semantics executable? How efficiently so? Can semantic definitions be tested to validate their agreement with an informal semantics?

- *Range of Applicability.* Can it be applied to programming languages and to software and hardware modeling languages? Can it naturally support nontrivial features such as concurrency and real time?

- *Scalability.* Can it be used in practice to give full definitions of real languages like Java or C? And of real software and hardware modeling languages?

- *Integrability.* How well can the semantics be integrated with language implementations and language analysis tools? Can it really be used as the *basis* on which such implementations and analysis tools are built?

This paper is a progress report on the efforts by various researchers in the rewriting logic semantics project to positively answer these questions. After summarizing some related work below, we give an overview of rewriting logic semantics in Section 2. Subsequent sections then describe in more detail: (i) modularity of definitions and the support for highly modular definitions provided by the $\mathbb{K}$ framework (Section 3); (ii) semantics of programming languages (Section 4); semantics of real-time language (Section 5); (iv) semantics of software modeling languages (Section 6); (v) semantics of hardware description languages (Section 7); (vi) abstract semantics and static analysis (Section 8); (vii) model checking verification (Section 9); and (viii) deductive verification (Section 10). We finish with some concluding remarks in Section 11.

This paper is a substantial extension of the conference paper [4], an extension in which we have treated several key topics in greater depth and have incorporated some more recent results. Specifically, a new section has been added discussing the $\mathbb{K}$ framework, its semantics and its implementation in more depth, namely Section 3.3. Since in the meanwhile the semantics of C has been completed, Section 4 now gives updated statistics as well as a detailed comparison with related work. Similarly, since our MatchC verifier and its underlying theory have been significantly advanced recently, Section 10 gives more detail on matching logic verification and enumerates several non-trivial verification efforts with MatchC. Likewise, our treatment of modeling language semantics and verification has been substantially extended by adding the following new sections: (i) Section 6.2, where the semantics of the real-time modeling languages Ptolemy II and Synchronous AADL is explained and illustrated with examples; (ii) Section 9.2 on model checking verification of Ptolemy II models; and (iii) Section 9.3 on model checking verification of Synchronous AADL models.

1.1. Related Work

There is much related work on frameworks for defining programming languages. Without trying to be exhaustive, we mention only some of them which are most closely related to rewriting logic, and point out some relationships to rewriting logic semantics (RLS). Frameworks like game semantics, monads, and nominal logics, although interesting and capable of defining executable and modular/compositional semantics of programming languages, are less related to rewriting logic and thus not discussed here.

Structural Operational Semantics (SOS). Several variants of structural operational semantics have been proposed. We refer to [3] for an in-depth comparison between SOS and RLS. A key point made in [3], and also made in Section 2.5, is that RLS is a framework supporting many different definitional styles. In particular, it can naturally and faithfully express many diffent SOS styles such as: small-step SOS [5], big-step SOS [6], MSOS [7], reduction semantics [8], continuation-based semantics [9], and the CHAM [10]. Compared to SOS, rewrite logic has several advantages, including: (i) the distinction between deterministic computation, expressed with equations, and concurrent computation, expressed with rules (see Section 2.4); and (ii) the seamless integration of denotational and operational semantics (see Section 2.3).

Algebraic denotational semantics. This approach, (see [11, 12, 13, 14] for early papers and [15, 16] for two more recent books), is the special case of RLS where the rewrite theory $\mathcal{R}_{\mathcal{L}}$ defining a language $\mathcal{L}$ is an equational theory. Its main limitation is that it is well suited for giving executable semantics to *deterministic* languages, but not that well suited for nondeterministic or concurrent language definitions. That's because programs in non-deterministic or concurrent languages can produce many different results, and an executable equational semantics of the language would end up inconsistently equating those different results, because the application of equations is reversible. Rewriting logic allows to use (irreversible) rewrite rules to define the non-deterministic features of a language, so the problem above is elegantly avoided.

Higher-order approaches. The most classic higher-order approach is *denotational semantics* [17, 18, 19, 20]. Denotational semantics has some similarities with its first-order algebraic cousin mentioned above, since both are based on semantic equations and both are best suited for deterministic languages. Higher-order functional languages or higher-order theorem provers can be used to give an executable semantics to programming languages, including the use of Scheme in [21], the use of ML in [22], and the use of Common LISP within the ACL2 prover in [23]. There is also a body of work on using monads [24, 25, 26] to implement language interpreters in higher-order functional languages; the monadic approach has better modularity characteristics than standard SOS. Some higher-order approaches are based on the use of higher-order abstract syntax (HOAS) [27, 28] and higher-order logical frameworks, such as LF [28] or λ-Prolog [29], to encode programming languages as formal logical systems; for a good example of recent work in this direction see [30] and references there. As in the case of algebraic denotational semantics, the main limitation of denotational semantics is that, being essentially a functional framework, it is not well suited for dealing with concurrency and non-determinism.

Logic-programming-based approaches. Going back to the Centaur project [31, 32], logic programming has been used as a framework for SOS language definitions. Note that λ-Prolog [29] belongs both in this category and in the higher-order one. For a recent textbook giving logic-programming-based language definitions, see [33]. In some ways, logic programming approaches have the opposite limitations of those in denotational semantics: there is no problem in dealing with non-determinism in logic programming; but functional computations are typically encoded in an awkward, relational way.

Abstract state machines. Abstract State Machine (ASM) [34] can encode any computation and have a rigorous semantics, so any programming language can be defined as an ASM and thus implicitly be given a semantics. Both big- and small-step ASM semantics have been investigated. The semantics of various programming languages, including Java [35], has been given using ASMs. In terms of executability and scalability to real languages, the ASM approach has clearly demonstrated that it can be applied effectively to large languages.

Other RLS work. RLS is a collective international project. There is by now a substantial body of work demonstrating the usefulness of this approach, e.g., [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 1, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63], and we describe some even more recent advances in this paper. A first snapshot of the RLS project was given in [1], a second in [2], and a third in [3], with this paper as the fourth snapshot.

2. Rewriting Logic Semantics in a Nutshell

Before describing in more detail the different advances in the rewriting logic semantics project we give here an overview of it. Be begin with a short summary of rewriting logic as a semantic framework for concurrent systems. Then we explain how it can be used to give both an operational and a denotational semantics to a programming language. Thanks to the distinction between equations and rules, this semantics can be given at various levels abstraction. Furthermore, a wide range of definitional styles can be naturally supported. We explain how rewriting logic semantics has been extended to: (i) real-time languages; (ii) software modeling languages; and (iii) hardware description languages. We finally explain how a rewriting logic semantics can be used for static analysis, and for model checking and deductive verification of programs.

2.1. Rewriting Logic

The goal of rewriting logic [64] is to provide a flexible logical framework to specify concurrent systems. A concurrent system is specified as a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory, and R is a set of (possibly conditional) rewrite rules. The equational theory (Σ, E) specifies the concurrent system's set of states as an algebraic data type, namely, as the initial algebra of the equational theory (Σ, E). Concretely, this means that a distributed state is mathematically represented as an E-equivalence class $[t]_E$ of terms built up with the operators declared in Σ, *modulo* provable equality using the equations E, so that two state representations t and t' describe the *same* state if and only if one can prove the equality $t = t'$ using the equations E.

The rules R specify the system's *local concurrent transitions*. Each rewrite rule in R has the form $t \to t'$, where t and t' are Σ-terms. The left-hand side t describes a *local firing pattern*, and the right-hand side t' describes a corresponding *replacement pattern*. That is, any fragment of a distributed state which is an instance of the firing pattern t can perform a local concurrent transition in which it is replaced by the corresponding instance of the replacement pattern t'. Both t and t' are typically *parametric* patterns, describing not single states, but parametric families of states. The parameters appearing in t and t' are precisely the *mathematical variables* that t and t' have, which can be instantiated to different concrete expressions by a *substitution*, that is, a mapping θ sending each variable x to a term $\theta(x)$. The instance of t by θ is then denoted $\theta(t)$.

The most basic *logical deduction steps* in a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ are precisely atomic concurrent transitions, corresponding to applying a rewrite rule $t \to t'$ in R to a state fragment which is an instance of the firing pattern t by some substitution θ. That is, up to E-equivalence, the state is of the form $C[\theta(t)]$, where C, called the *context*, is the rest of the state not affected by this atomic transition. Then, the resulting state is precisely $C[\theta(t')]$, so that the atomic transition has the form $C[\theta(t)] \to C[\theta(t')]$. Rewriting is *intrinsically concurrent*, because many other atomic rewrites can potentially take place in the context C (and in the substitution θ), at the same time that the local atomic transition $\theta(t) \to \theta(t')$ happens. The rules of deduction of rewriting logic [64, 65] (which in general allow rules in R to be *conditional*) precisely describe all the possible, complex concurrent transitions that a system can perform, so that concurrent computation and logical deduction *coincide*.

2.2. Defining Programming Languages

The flexibility of rewriting logic to naturally express many different models of concurrency can be exploited to give *formal definitions of concurrent programming languages* by specifying the concurrent model of a language $\mathcal{L}$ as a rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$, where: (i) the signature $\Sigma_{\mathcal{L}}$ specifies both the syntax of $\mathcal{L}$ and the types and operators needed to specify semantic entities such as the store, the environment, input-output, and so on; (ii) the equations $E_{\mathcal{L}}$ can be used to give semantic definitions for the *deterministic* features of $\mathcal{L}$ (a sequential language typically has only deterministic features and can be specified just equationally as $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$); and (iii) the rewrite rules $R_{\mathcal{L}}$ are used to give semantic definitions for the concurrent features of $\mathcal{L}$ such as, for example, the semantics of threads.

By specifying the rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ in a rewriting logic language like Maude[1] [66], it becomes not just a mathematical definition but an *executable* one, that is, an *interpreter* for $\mathcal{L}$. Furthermore, one can leverage Maude's generic search and LTL model checking features to automatically endow $\mathcal{L}$ with powerful *program analysis capabilities.* For example, Maude's search command can be used in the module $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ to detect any violations of invariants, e.g., a deadlock or some other undesired state, of a program in $\mathcal{L}$. Likewise, for terminating concurrent programs in $\mathcal{L}$ one can model check any desired LTL property. All this can be effectively done not just for toy languages, but for real ones such as Java and the JVM, Scheme, and C (see Section 4 for a discussion of such real-language applications), and with performance that compares favorably with state-of-the-art model checking tools for real languages. As we show in Section 8, a wide variety of static analysis formal tools can also be automatically obtained from suitable *abstract* rewriting logic semantics of $\mathcal{L}$. Finally, using a deductive approach like matching logic (see Section 10), which is directly based on the rewriting logic semantics $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ of $\mathcal{L}$, it is also possible to obtain a highly effective deductive verification tool for programs in $\mathcal{L}$.

2.3. Operational vs. Denotational Semantics

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has both a *deduction-based operational semantics*, and an *initial model denotational semantics.* Both semantics are defined naturally out of the proof theory of rewriting logic [64, 65]. The deduction-based operational semantics of $\mathcal{R}$ is defined as the collection of *proof terms* [64] of the form $\alpha : t \longrightarrow t'$. A proof term α is an algebraic description of a proof tree proving $\mathcal{R} \vdash t \longrightarrow t'$ by means of the inference rules of rewriting logic. What

[1] Other rewriting logic languages, such as ELAN or CafeOBJ, can likewise be used. Maude has the advantage of efficiently supporting not only execution, but also linear temporal logic (LTL) model checking verification.

such proof trees describe are the different *finitary concurrent computations* of the concurrent system axiomatized by $\mathcal{R}$.

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has also a *model-theoretic semantics*, so that the inference rules of rewriting logic are sound and complete with respect to satisfaction in the class of models of $\mathcal{R}$ [64, 65]. Such models are *categories* with a (Σ, E)-algebra structure [64]. These are "true concurrency" denotational models of the concurrent system axiomatized by $\mathcal{R}$. That is, this model theory gives a precise mathematical answer to the question: when do two descriptions of two concurrent computations denote *the same* concurrent computation? The class of models of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has an *initial model* $\mathcal{T}_{\mathcal{R}}$ [64]. The initial model semantics is obtained as a *quotient* of the just-mentioned deduction-based operational semantics, precisely by axiomatizing algebraically when two proof terms $\alpha : t \longrightarrow t'$ and $\beta : u \longrightarrow u'$ denote the same concurrent computation.

In particular, if a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ specifies the semantics of a concurrent programming language $\mathcal{L}$, its denotational semantics is given by the initial model $\mathcal{T}_{\mathcal{R}_{\mathcal{L}}}$, and its operational semantics is given by the proof terms built by the rewriting deduction. As we explain below, many different styles of operational semantics, including various SOS styles, can be naturally obtained as special instances of this general, logic-based operational semantics.

2.4. The Abstraction Dial

Unlike formalisms like SOS, where there is only one type of semantic rule, rewriting logic semantics provides a key distinction between *deterministic rules*, axiomatized by equations, and concurrent and typically *non-deterministic* rules, axiomatized by non-equational rules. More precisely, for the rewriting logic semantics $\mathcal{R}_{\mathcal{L}}$ of a language $\mathcal{L}$ to have good executabity properties, we require $\mathcal{R}_{\mathcal{L}}$ to be of the form $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \cup B_{\mathcal{L}}, R_{\mathcal{L}})$, where: (i) $B_{\mathcal{L}}$ is a collection of *structural axioms*, such as associativity and/or commutativity, and/or identity of certain operators in $\Sigma_{\mathcal{L}}$; (ii) the equations $E_{\mathcal{L}}$ are *confluent modulo* the structural axioms $B_{\mathcal{L}}$; and (iii) the rules $R_{\mathcal{L}}$ are *coherent* with the equations $E_{\mathcal{L}}$ modulo the structural axioms $B_{\mathcal{L}}$ [67]. Conditions (i)–(iii) make $\mathcal{R}_{\mathcal{L}}$ *executable*, so that using a rewriting logic language like Maude we automatically get an interpreter for $\mathcal{L}$.

As already mentioned, what the equations $E_{\mathcal{L}}$ axiomatize are the *deterministic features* of $\mathcal{L}$. Instead, the truly concurrent features of $\mathcal{L}$ are axiomatized by the non-equational rules $R_{\mathcal{L}}$. The assumption of determinism is precisely captured by $E_{\mathcal{L}}$ being a set of *confluent equations* (modulo $B_{\mathcal{L}}$), so that their evaluation, if terminating, has a *unique* final result.

All this means that rewriting logic comes with a built-in "abstraction dial." The least abstract possible position for such a dial is to turn the equations $E_{\mathcal{L}}$ into rules, yielding the theory $(\Sigma_{\mathcal{L}}, B_{\mathcal{L}}, E_{\mathcal{L}} \cup R_{\mathcal{L}})$; this is typically the approach taken by SOS definitions. The specification $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \cup B_{\mathcal{L}}, R_{\mathcal{L}})$ can already achieve an enormous abstraction, which typically makes the difference between tractable and intractable model checking analysis. The point is that the equations $E_{\mathcal{L}}$ now identify all intermediate execution states obtained by deterministic steps, yielding a typically enormous state space reduction. Sometimes we may be able to turn the dial to an *even more abstract position* by further decomposing $R_{\mathcal{L}}$ as a disjoint union $R_{\mathcal{L}} = R'_{\mathcal{L}} \cup G_{\mathcal{L}}$, so that the rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \cup G_{\mathcal{L}} \cup B_{\mathcal{L}}, R'_{\mathcal{L}})$ still satisfies conditions (i)–(iii). That is, we may be able to identify rules $G_{\mathcal{L}}$ describing concurrent executions which, by being confluent, can be turned into equations. For example, for $\mathcal{L} = Java$, the JavaFAN rewriting logic semantics of Java developed by the late Feng Chen turns the abstraction dial as far as possible, obtaining a set E_{Java} with hundreds of equations, and a set R_{Java} with just 5 rules. This enormous state space reduction is a key reason why the JavaFAN model checker compares favorably with other state-of-the-art Java model checkers [46].

But the abstraction story does not end here. After all, the semantics $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \cup G_{\mathcal{L}} \cup B_{\mathcal{L}}, R'_{\mathcal{L}})$ obtained by turning the abstraction dial as much as possible is still a *concrete* semantics. We might call it "the most abstract concrete semantics possible." For many different static analysis purposes one wants to take a further abstraction step, which further collapses the set of states by defining a suitable *abstract semantics* for a language $\mathcal{L}$. The point is that, instead of a "concrete semantics" describing the actual execution of programs in $\mathcal{L}$, one can just as easily define an "abstract semantics" $(\Sigma^A_{\mathcal{L}}, E^A_{\mathcal{L}}, R^A_{\mathcal{L}})$ describing any desired abstraction A of $\mathcal{L}$. A good example is type checking, where the values manipulated by the abstract semantics are the types. All this means that many different forms of program analysis, much more scalable than model checking based on a language's concrete semantics, become available essentially for free by using a tool like Maude to execute and analyze one's desired abstract semantics $(\Sigma^A_{\mathcal{L}}, E^A_{\mathcal{L}}, R^A_{\mathcal{L}})$. This is further discussed in Section 8.

2.5. An Ecumenical Movement

For purposes of formally defining the semantics of a programming language, rewriting logic should be viewed not as a competitor to other approaches, but as an "ecumenical movement" providing a framework where many different definitional styles can happily coexist. From its early stages rewriting logic has been recognized as ideally suited for SOS definitions [68, 69], and has been used to give SOS definitions of programming languages in quite different styles, e.g., [41, 48, 70, 44, 46, 45]. What the paper [3] makes explicit is both the wide range of SOS styles supported, and the possibility of defining new styles that may have specific advantages over traditional ones. Indeed, the intrinsic flexibility of rewriting logic means that it does not prescribe a fixed style for giving semantic definitions. Instead, *many different styles* such as, for example, small-step or big-step semantics, reduction semantics, CHAM-style semantics, modular structural operational semantics, or continuation semantics, can all be naturally supported [3]. But not all styles are equally efficient; for example, small-step semantics makes heavy use of conditional rewrite rules, insists on modeling every single computation step as a rule in $R_{\mathcal{L}}$, and is in practice horribly inefficient. Instead, the continuation semantics style described in [3] and used in, e.g., [46] is very efficient. Furthermore, as already mentioned, the distinction between equations and rules provides an "abstraction dial" not available in some definitional styles but enormously useful for state space reduction purposes. Of particular interest are *modular* definitional styles, which are further discussed in Section 3.

2.6. Defining Real-Time Languages

In rewriting logic, real-time systems are specified with *real-time rewrite theories* [71]. These are just ordinary rewrite theories $\mathcal{R} = (\Sigma, E \cup B, R)$ such that: (i) there is a sort *Time* in Σ such that (Σ, E) contains an algebraic axiomatization of a time data type, where time can be either discrete or continuous; (ii) there is also a sort *GlobalState*, where terms of sort *GlobalState* are pairs (t, r), with t an "untimed" or "discrete" state (which may however contain continuous, time-related quantities such as timers), and r is a term of sort *Time* (that is, the global state is a discrete state plus a global clock); and (iii) the rules R are either: (a) *instantaneous* rules, which do not change the time and only rewrite the discrete part of the state, or (b) *tick* rules, of the form

$$(t, r) \to (t', r') \;\; \mathit{if} \;\; C$$

where t and t' are term patterns describing discrete states, r and r' are terms of sort *Time*, and C is the rule's condition. That is, tick rules advance the global clock and also update the discrete state to reflect the passage of time (for example, timers may be decreased, and so on). Real-Time rewrite theories provide a very expressive semantic framework in which many models of real-time systems can be naturally expressed [71]. The Real-Time Maude language [72] is an extension of Maude that supports specification, simulation, and model checking analysis of real-time systems specified as real-time rewrite theories.

How should the formal semantics of a *real-time* programming language be defined? And how can programs in such a language be formally analyzed? The obvious RLS answers are: (i) "with a real-time rewrite theory," and (ii) "by real-time model checking and/or deductive reasoning based on such a theory." Of course, the effectiveness of these answers has to be shown in actual languages. This is done in Sections 5 and 6.

More generally, real-time systems can also be *probabilistic*, and can be modeled by probabilistic rewrite theories [73, 74]. In the analysis of such systems, which include, among others, distributed stochastic hybrid systems naturally modeled as rewrite theories [75], analysis of *quantitative properties*, yielding as result a numeric value and not just a "true" or "fasle" answer, is of great interest. Such quantitative formal analysis can be performed in the QuaTex quantitative temporal logic [73], and can be analyzed in parallel for Maude specifications using the PVesta tool [76]. We refer the reader to [77, 78, 75, 79, 80] for examples of various real-time distributed and probabilistic software architectures which have been specified and analyzed in rewrite logic in this way. In Sections 6.1 and 6.2 we focus on the rewrite logic semantics modeling languages for *embedded systems* such as AADL (`http:www.aadl.info`) and Ptolemy [81]. Although the semantics given in Section 6.2 focuses only on the real-time aspects, it can be naturally extended to include probabilistic aspects due, for example, to component failures, or to interactions with an unpredictable environment.

2.7. Defining Modeling Languages

It is well known that the most expensive errors in system development are not coding errors but design errors. Since design errors affect the overall structure of a system and are often discovered quite late in the development cycle, they can be enormously expensive to fix. All this is uncontroversial: there is widely-held agreement that, to develop systems, designs themselves should be made machine-representable, and that tools are needed to keep such designs consistent and to uncover design errors as early as possible. This has led to the development of many software modeling languages.

There are however two main limitations at present. The first is that some of these modeling notations lack a formal semantics: they can and do mean different things to different people. The second is that this lack of semantics manifests itself at the practical level as a lack of *analytic power*, that is, as an incapacity to uncover expensive design errors which could have been caught by better analysis. It is of course virtually impossible to solve the second problem without solving the first: without a precise mathematical semantics any analytic claims about satisfaction of formal requirements are meaningless.

The practical upshot of all this is that a semantic framework such as rewriting logic can play an important role in: (i) giving a precise semantics to modeling languages; and in (ii) endowing such languages and notations with powerful formal analysis capabilities. Essentially the approach is the same as for programming languages. If, say, $\mathcal{M}$ is a modeling language, then its formal semantics will be a rewrite theory of the form $(\Sigma_{\mathcal{M}}, E_{\mathcal{M}}, R_{\mathcal{M}})$. If the modeling language $\mathcal{M}$ provides enough information about the dynamic behavior of models, the equations $E_{\mathcal{M}}$ and the rules $R_{\mathcal{M}}$ will make $\mathcal{M}$ *executable*, that is, it will be possible to *simulate* models in $\mathcal{M}$ before they are realized by concrete programs, and of course such models thus become amenable to various forms of *formal analysis*. All these ideas are further discussed in Section 6.

2.8. Defining Hardware Description Languages

What is hardware? What is software? It depends in part on the level of abstraction chosen, and on specific implementation decisions: a given functionality may sometimes be realized as microcode, other times as code running on an FPGA, and yet other times may be implemented in custom VLSI. All this means that the difference between the semantics of digital hardware in some Hardware Description Language (HDL), and that of a programming language is not an essential one, just one about which level of abstraction is chosen. From the point of view of rewriting logic, both the semantics of an HDL and that of a programming language can be expressed by suitable rewrite theories. We further discuss the rewriting logic semantics of HDLs in Section 7.

2.9. Formal Analysis Methods and Tools

The fact that, under simple conditions, rewriting logic specifications are executable, means that the rewriting logic semantics of a language, whether a programming language, or a modeling language, or an HDL, is *executable* and therefore yields an *interpreter* for the given language when run on a rewriting logic system such as Maude. Since the language in question may not have any other formal semantics, the issue of whether the semantic definitions correctly capture the language's informal semantics is a nontrivial matter; certainly not trivial at all for real languages which may require hundreds of semantic rules. The fact that the semantics is executable is very useful in this regard, since one can *test* the correctness of the definitions by comparing the results from evaluating programs in the interpreter obtained from the rewriting logic semantics and in an actual language implementation. The usefulness of this approach is further discussed for the case of the semantics of C in Section 4.

Once the language specifier is sufficiently convinced that his/her semantic definitions correctly capture the language's informal semantics, various sophisticated forms of program analysis become possible. If some abstract semantics for the language in question has been defined, then the abstract semantic definition can be directly used as *static analysis tool.* Since various abstract semantics may be defined for diverse analysis purposes, a collection of such tools may be developed. We further discuss this idea in Section 8.

Using a tool like Maude, the concrete rewriting logic semantics of a language becomes not just an interpreter, but also a *model checker* for the language in question. The point is that Maude can model check properties for any user-specified rewrite theory. Specifically, it can perform reachability analysis to detect violations of invariants using its breadth-first search feature; and it can also model check temporal logic properties with its LTL model checker. Such features can then be used to model check programs in the language whose rewriting semantics one has defined, or in an abstraction of it, as explained in Section 9.

Static analysis and model checking do not exhaust the formal analysis possibilities. A language's rewriting logic semantics can also be used as the basis for *deductive reasoning* about programs in such a language. The advantage of directly basing deductive reasonign methods on the semantics is that there is no gap between the operational semantics and the "program logic." This approach has been pioneered by *matching logic* [82, 83, 84, 85, 86], a program verification logic, with substantial advantages over both Hoare logic and separation logic, which uses a language's rewriting logic semantics, including the possibility of using patterns to symbolically characterize sets of states, to mechanize the formal verification of programs, including programs that manipulate complex data structures. More on matching logic and the MatchC tool in Section 10.

All the above are strong arguments in favor of an executable formal semantics of a programming language, as opposed to an implementation. A hasty reader may think that there is no fundamental difference between a formal executable semantics and an implementation of a programming language, because an implementation can also be framed as an executable semantics, possibly going through a given and fixed semantics of the implementation language, while a formal executable semantics must provide all the implementation details in order to be executable. There are, however, at least two important factors that the above considerations ignore and that one should consider.

First, a formal rewrite logic semantics captures directly each language features in a natural and modular way, which substantially eases reasoning about programs in the defined language. For example, consider an "implementation" of a programming language on a Turing machine. While Turing machines have crystal clear semantics, which thus indirectly yield formal semantics to the implemented language, the user of such a semantics would have to reason in terms of Turing machine concepts, which may sometimes be quite far from the actual programming language concepts that one wants to reason about (in fact, the language concepts are typically "compiled away" at the Turing machine level). In contrast, our formal reasoning techniques for rewrite logic semantics allow us to reason directly about the defined language, without any encoding or translation.

Second, the "implementation details" of a formal executable semantics are considerably more abstract and, what is crucial, are *mathematical objects* which can be *directly* used for formal reasoning. In contrast, the implementation details of a compiler or an interpreter are *not* in any sense mathematical objects: one would first need a formal semantics of the language in which such a compiler is written, and this would create a considerable gap when reasoning about a high-level program. To give a simple example, a program environment may be implemented as a hash table in a compiler or an interpreter for performance reasons, while it typically is a formally defined finite-domain map algebraic data-type in an executable semantics, a mathematical object with which we can directly reason about environments.

3. Modular Definitions and the $\mathbb{K}$ Framework

One major impediment blocking the broader use of semantic frameworks is the lack of scalability of semantic definitions. Lack of *modularity* is one of the main causes for this lack of scalability. Indeed, in many frameworks one often needs to redefine the semantics of the existing language features in order to include new, unrelated features. For example, in conventional SOS [5] one needs to more than double the number of rules in order to include an abrupt termination construct to a language, because the termination "signal" needs to be propagated through all the language constructs. Mosses' Modular SOS (MSOS) [7] addresses the non-modularity of SOS; it has been shown that MSOS can be faithfully represented in rewriting logic, in a way that also preserves its modularity [36, 49, 48, 87, 88]. We here report on the $\mathbb{K}$ framework, developed in parallel with the MSOS approach.

$\mathbb{K}$ [89] is a modular executable semantic framework derived from rewriting logic. It works with terms, but its concurrent semantics is best explained in terms of graph rewriting intuitions [90, 91]. $\mathbb{K}$ was first introduced by the second author in the lecture notes of a programming language design course at the University of Illinois at Urbana-Champaign (UIUC) in Fall 2003 [92], as a means to modularly define concurrent languages in rewriting logic using Maude. Programming languages, calculi, as well as type systems or formal analyzers can be defined in $\mathbb{K}$ by making use of special, potentially nested *cell* structures, and

Original lang. syntax	K Strict.	K Semantics
$\mathit{AExp} ::= \mathit{Int}$		$\langle \underset{i}{\underline{x}}\ \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto i\ \cdots\rangle_{\mathsf{state}}$
$\mid\ \mathit{Id}$		
$\mid\ \mathit{AExp} \texttt{ + } \mathit{AExp}$	[*strict*]	$i_1 \texttt{ + } i_2 \to i_1 +_{\mathit{Int}} i_2$
$\mid\ \mathit{AExp} \texttt{ / } \mathit{AExp}$	[*strict*]	$i_1 \texttt{ / } i_2 \to i_1 /_{\mathit{Int}} i_2$ where $i_2 \neq 0$
$\mathit{BExp} ::= \mathit{Bool}$		
$\mid\ \mathit{AExp} \texttt{ <= } \mathit{AExp}$	[*seqstrict*]	$i_1 \texttt{ <= } i_2 \to i_1 \leq_{\mathit{Int}} i_2$
$\mid\ \texttt{not } \mathit{BExp}$	[*strict*]	$\texttt{not } t \to \neg_{\mathit{Bool}}\ t$
$\mid\ \mathit{BExp} \texttt{ and } \mathit{BExp}$	[*strict*(1)]	$\mathit{true} \texttt{ and } b \to b$
		$\mathit{false} \texttt{ and } b \to \texttt{false}$
$\mathit{Stmt} ::= \texttt{skip}$		$\texttt{skip} \to \cdot$
$\mid\ \mathit{Id} \texttt{ := } \mathit{AExp}$	[*strict*(2)]	$\langle \underset{\cdot}{\underline{x \texttt{ := } i}}\ \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto \underset{i}{\underline{\_}}\ \cdots\rangle_{\mathsf{state}}$
$\mid\ \mathit{Stmt} \texttt{ ; } \mathit{Stmt}$		$s_1 \texttt{ ; } s_2 \rightharpoonup s_1 \curvearrowright s_2$
$\mid \texttt{if } \mathit{BExp}$	[*strict*(1)]	$\texttt{if } \mathit{true} \texttt{ then } s_1 \texttt{ else } s_2 \to s_1$
$\texttt{then } \mathit{Stmt} \texttt{ else } \mathit{Stmt}$		$\texttt{if } \mathit{false} \texttt{ then } s_1 \texttt{ else } s_2 \to s_2$
$\mid \texttt{while } \mathit{BExp} \texttt{ do } \mathit{Stmt}$		$\langle \underset{\texttt{if } b \texttt{ then } (s \texttt{ ; while } b \texttt{ do } s) \texttt{ else } \cdot}{\underline{\texttt{while } b \texttt{ do } s}}\ \cdots\rangle_{\mathsf{k}}$
$\mathit{Pgm} ::= \texttt{var } \mathbf{List\{}\mathit{Id}\mathbf{\}} \texttt{ ; }$ Stmt		$\langle \underset{s}{\underline{\texttt{var } xl \texttt{ ; } s}}\rangle_{\mathsf{k}}\ \langle \underset{xl \mapsto 0}{\underline{\cdot}}\rangle_{\mathsf{state}}$

Figure 1: $\mathbb{K}$ definition of IMP: syntax (left), annotations (middle) and semantics (right); $x \in \mathit{Id}$, $xl \in \mathbf{List\{}\mathit{Id}\mathbf{\}}$, $i, i_1, i_2 \in \mathit{Int}$, $t \in \mathit{Bool}$, $b \in \mathit{BExp}$, $s, s_1, s_2 \in \mathit{Stmt}$

rules. There are two types of $\mathbb{K}$ rules: *computational rules*, which count as computational steps, and *structural rules* (or "half equations"), which do not count as computational steps. The role of the structural rules is to rearrange the term so that the computational rules can apply. $\mathbb{K}$ rules are *unconditional* (they may have side conditions, though), and they are *context-insensitive*. $\mathbb{K}$ rules are regarded as *transactions*, stating what is read-only, what is read-write, and what is irrelevant. This allows for true concurrency even in the presence of sharing.

We introduce $\mathbb{K}$ by means of a simple imperative language, called IMP. In Section 3.2 we extend IMP with several features (including dynamic threads) into IMP++, and in Section 3.3 we give some details about the $\mathbb{K}$ semantics and its current implementation. Later, in Section 8.1, we show how one can use $\mathbb{K}$ to define a type checker for IMP++. This language experiment is borrowed from [89], where more details about $\mathbb{K}$ can be found. We refer the interested reader to `http://k-framework.org` for papers, and implementation of a $\mathbb{K}$ tool, as well as for many language definitions in $\mathbb{K}$ following different paradigms, including object-oriented languages, functional languages, and logic programming languages.

3.1. $\mathbb{K}$ *Semantics of* IMP

Figure 1 shows the complete $\mathbb{K}$ definition of IMP, except for the configuration (explained below). The left column gives the IMP syntax. The middle column augments it with $\mathbb{K}$ *strictness attributes*, stating the evaluation strategy of some language constructs. Finally, the right column gives the semantic rules.

Language syntax is typically defined in $\mathbb{K}$ using an "algebraic" context-free notation, i.e., one which allows users to make use of list, set, multi-set and map structures without defining them. Note, e.g., that we used $\mathbf{List\{}\mathit{Id}\mathbf{\}}$ as a non-terminal in the syntax of IMP in Figure 1. System configurations are defined in the same style. Configurations in $\mathbb{K}$ are organized as potentially nested structures of *cells*, which are typically labeled to distinguish them from each other. We use angle brackets as cell wrappers. The $\mathbb{K}$ configuration of IMP can be defined as:

$$\mathit{Configuration}_{\mathrm{IMP}} \quad \equiv \quad \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathbf{Map\{}\mathit{Id} \mapsto \mathit{Int}\mathbf{\}}\rangle_{\mathsf{state}}\rangle_{\top}$$

Same like for $\mathbf{List\{...\}}$, we use $\mathbf{Map\{}S_1 \mapsto S_2\mathbf{\}}$ as a non-terminal corresponding to finite-domain maps from elements of sort S_1 to elements of sort S_2; such maps are syntactically represented as (space- or comma- separated) sequences of pairs $t_1 \mapsto t_2$, with t_1 a term of sort S_1 and t_2 a term of sort S_2. In words, IMP configurations consist of a top cell $\langle\ldots\rangle_{\top}$ containing two other cells inside: a cell $\langle\ldots\rangle_{\mathsf{k}}$ which holds a term of sort K (the computation) and a cell $\langle\ldots\rangle_{\mathsf{state}}$ which holds a map from variables to integers. As examples of IMP configurations, $\langle\langle \texttt{x := 1; y := x+1}\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{state}}\rangle_{\top}$ is a configuration holding program "`x := 1; y := x+1`" and empty state, and $\langle\langle \texttt{x := 1; y := x+1}\rangle_{\mathsf{k}}\ \langle \texttt{x} \mapsto 0\ \ \texttt{y} \mapsto 1\rangle_{\mathsf{state}}\rangle_{\top}$ is one holding the same program and a state with bindings $\texttt{x} \mapsto 0$ and $\texttt{y} \mapsto 1$.

The sort K, for *computational structures* or simply *computations*, has a special meaning in $\mathbb{K}$. The intuition for terms of sort K is that they have computational meaning, such as programs or program fragments have. Formally, computations extend the syntax of the original language (i.e., all syntactic categories are sunk into K) with a list structure with "$\curvearrowright$" (read "followed by") as binary concatenation of computations and with "$\cdot$" as the empty computation. For example, the intuition for a computation of the form $T_1 \curvearrowright T_2 \curvearrowright \cdots \curvearrowright T_n$ is that the enlisted (computational) tasks should be processed sequentially.

Computations give a general and uniform means to define and handle evaluation strategies of language constructs. For example, evaluation contexts and/or (first-order) continuations can be regarded as computations: "$v \curvearrowright c$" can be thought of as "$c[v]$, that is, evaluation context c applied to v" or as "passing v to continuation c". In fact, $\mathbb{K}$ allows one to define evaluation strategies over the language syntax both directly, by means of rules over computations, or indirectly, by means of *strictness* attributes like in the middle column in Figure 1. However, the strictness attributes are nothing but convenient *notations*, which desugar into rules. For example, the evaluation strategies of sum, comparison and conditional in IMP specified by the strictness attributes in Figure 1 can be defined using the following *structural rules* (for diversity, we assume that the sum `+` evaluates its arguments non-deterministically and the comparison `<=`

evaluates its arguments sequentially):

$$\begin{array}{rcl}
a_1 \texttt{ + } a_2 & \rightleftharpoons & a_1 \curvearrowright \square \texttt{ + } a_2 \\
a_1 \texttt{ + } a_2 & \rightleftharpoons & a_2 \curvearrowright a_1 \texttt{ + } \square \\
a_1 \texttt{ <= } a_2 & \rightleftharpoons & a_1 \curvearrowright \square \texttt{ <= } a_2 \\
i_1 \texttt{ <= } a_2 & \rightleftharpoons & a_2 \curvearrowright i_1 \texttt{ <= } \square \\
\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 & \rightleftharpoons & b \curvearrowright \texttt{if } \square \texttt{ then } s_1 \texttt{ else } s_2
\end{array}$$

The symbol $\rightleftharpoons$ stands for two structural rules, one left-to-right and another right-to-left. Inspired from chemical abstract machine terminology [10], we informally call the left-to-right rules above *heating rules*, with the expression passed in front of the computation the *hot* one, and the right-to-left rules *cooling rules.*

As discussed shortly, not all structural rules in a $\mathbb{K}$ definition are reversible, although those corresponding to evaluation strategies like above typically are. The right-hand sides of the structural rules above contain, besides the task sequentialization operator $\curvearrowright$, *freezer* operators containing $\square$ in their names, such as $\square \texttt{+} \_$, $\_ \texttt{+} \square$, etc. The first rule above says that in any expression of the form $a_1 \texttt{+} a_2$, a_1 can be scheduled for processing while a_2 is being held for future processing. Since these rules are bi-directional, they can be used at will to structurally re-arrange the computations. Thus, when iteratively applied from left-to-right they fulfill the role of *splitting* syntax into an *evaluation context* (the tail of the resulting sequence of computational tasks) and a *redex* (the head of the resulting sequence), and when applied right-to-left they fulfill the role of *plugging* syntax into context. Our current implementation of $\mathbb{K}$ automatically generates rules like the above, plus heuristics to apply them in one direction or the other, from strictness annotations to syntax like in Figure 1 (middle column).

Structural rules like those above decompose and eventually push the tasks that are ready for processing to the top (or the left) of the computation. Semantic rules then tell how to process the atomic tasks. The right column in Figure 1 shows the $\mathbb{K}$ semantic rules of IMP. To explain them, let us first discuss the important notion of a $\mathbb{K}$ *rule*, which is a strict generalization of the usual notion of a rewrite rule. $\mathbb{K}$ rules explicitly mention the parts of the term that they read, write, or don't care about. The underlined parts are those which are written by the rule; the term underneath the line is the new subterm replacing the one above the line. All writes in a $\mathbb{K}$ rule are applied in *one parallel step*, and, with some reasonable restrictions discussed in [93, 91] that avoid read/write and write/write conflicts, writes in multiple $\mathbb{K}$ rule instances can also apply in parallel. The elipses " $\cdots$ " represent the volatile part of the term, that is, that part that the current rule does not care about and, consequently, can be concurrently modified by other rules. The operations which are not underlined represent the read-only part of the term: they need to stay unchanged during the application of the rule. For example, consider the assignment rule in Figure 1:

$$\langle \underset{\cdot}{\underline{x \texttt{ := } i}} \cdots\rangle_{\mathsf{k}} \; \langle \cdots\, x \mapsto \underset{i}{\underline{\_}} \cdots\rangle_{\mathsf{state}}$$

Original language syntax	K Strictness	K Semantics
$AExp\ ::= \ldots \mid \texttt{++}\, Id$		$\langle \frac{\texttt{++}\, x}{i +_{Int} 1} \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto \frac{i}{i +_{Int} 1} \cdots\rangle_{\mathsf{state}}$
$Stmt\ ::= \ldots$		
$\mid\ \texttt{print}\, AExp$	$[strict]$	$\langle \frac{\texttt{print}\, i}{\cdot} \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ \frac{\cdot}{i}\rangle_{\mathsf{output}}$
$\mid\ \texttt{halt}$		$\langle \frac{\texttt{halt} \curvearrowright \_}{\cdot}\rangle_{\mathsf{k}}$
$\mid\ \texttt{spawn}\, Stmt$		$\langle \frac{\texttt{spawn}\, s}{\cdot} \cdots\rangle_{\mathsf{k}}\ \frac{\cdot}{\langle s \curvearrowright die\rangle_{\mathsf{k}}}$
$K\ ::= \ldots \mid die$		$\langle die\rangle_{\mathsf{k}} \rightharpoondown \cdot$

Figure 2: K definition of IMP++ (extends that in Figure 1, *without changing anything*)

It says that once the assignment x `:=` i reaches the top of the computation, the value of x in the store is replaced by i and the assignment dissolves; in $\mathbb{K}$, "_" is a nameless variable of any sort and "$\cdot$" is the unit (or empty) computation ("$\cdot$" is a polymorphic unit of all list, set and multi-set structures). The rule for variable declarations in Figure 1 (last one) expects an empty state and allocates and initializes with 0 all the declared variables; the dotted or dashed lines signify that the rule is structural, which is discussed next.

$\mathbb{K}$ rules are split in two categories: *computational* and *structural*. Computational rules capture the intuition of computational steps in the execution of the defined system or language, while structural rules capture the intuition of structural rearrangement, rather than computational evolution, of the system. We use dashed or dotted lines in the structural rules. Ordinary rewrite rules are particular $\mathbb{K}$ rules, where the entire term pattern is replaced; for such rules we keep the standard notation $l \rightarrow r$ as syntactic sugar for computational rules, whereas the notation $l \rightharpoondown r$ or $l \rightharpoonup r$ is used as syntactic sugar for structural rules. Figure 1 shows three explicit structural rules (as already discussed, the strictness attributes correspond to implicit ones): s_1 **;** s_2 is rearranged as $s_1 \curvearrowright s_2$, loops are unrolled when they reach the top of the computation (unconstrained unrolling leads to non-termination), and declared variables are allocated in the state. Note that, unlike the implicit structural rules corresponding to evaluation strategies, these structural rules are not bi-directional.

3.2. Extending IMP

In this section we highlight the modularity of $\mathbb{K}$ by extending the IMP language in Section 3.1 with variable increment and dynamic threads. Figure 2 shows how the $\mathbb{K}$ semantics of IMP is seamlessly extended into a semantics for IMP++. To accommodate the output, a new cell needs to be added to the configuration:

$$Configuration_{\mathrm{IMP++}} \quad \equiv \quad \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathbf{Map\{}Id \mapsto Int\mathbf{\}}\rangle_{\mathsf{state}}\ \langle \mathbf{List\{}Int\mathbf{\}}\rangle_{\mathsf{output}}\ \rangle_{\top}$$

However, note that none of the existing IMP rules needs to change, because each of them only matches what it needs from the configuration. The construct `print` is strict and its rule adds the value of its argument to the end of the output buffer (matches and replaces the unit "$\cdot$" at the end of the buffer). The rule for `halt` dissolves the entire computation, and the rule for `spawn` creates a new $\langle\ldots\rangle_{\mathsf{k}}$ cell wrapping the spawned statement. The code in this new cell will be processed concurrently with the other threads. The last rule "cools down" a terminated thread by simply dissolving it; it is a structural rule since, again, we do not want it to count as a computational step.

Note that it is not always the case that a language extension only requires adding new cells to a configuration. Some extensions may need to restructure the semantic information in the configuration. For example, to add blocks and local variables and have the spawned threads share their parents' environments, we would need to split the state cell into a thread-local environment and a shared store, and then have a cell associated to each thread holding a computation cell and an environment. Many of the languages that come with the $\mathbb{K}$ tool distribution are defined this way (we encourage the interested reader to check them out at `http://k-framework.org`).

3.3. $\mathbb{K}$ *Semantics and Implementation*

In this section we give more details about the semantics and the implementation of $\mathbb{K}$, making an effort to separate the (easy) notational conventions, which can be mechanically desugared, from the actual semantic novelties of $\mathbb{K}$. The current implementation of the $\mathbb{K}$ framework, which we call the $\mathbb{K}$ *tool*, consists of a translator to Maude, which is implemented using Perl scripting (about 6,000 lines), Haskell (about 1,500 lines), and Maude (about 9,000 lines), and is available for download at the URL above. We will also explain theoretical trade-offs that the current implementation makes in order to achieve simplicity and higher performance.

Language syntax and configuration declarations like the ones illustrated above, as well as additional syntax that may be needed for defining needed semantic domains, are in the end nothing but programming-language-specific *notations*. Indeed, we incrementally learned that the language designers using $\mathbb{K}$ find it much more convenient to define syntax using context-free grammars (CFGs), since programming language manuals typically formalize syntax using such a notation. Also, defining the configuration in one place as a nested structure of cells specifying on the spot what each cell holds is more compact and intuitive to them than giving an algebraic signature. Nevertheless, all these notions and notations can be expressed as order-sorted algebraic specifications. In fact, our current implementation of $\mathbb{K}$ translates them mechanically into Maude algebraic specifications.

To translate language syntax defined using *context-free grammars* (CFGs), the $\mathbb{K}$ tool follows the well-known [94, 66] correspondence between CFGs and algebraic signatures written using the mixfix notation (i.e., operation names include underscores "_" as argument placeholders), which Maude supports. For example, giving the CFG production

$$\mathit{Stmt} ::= \texttt{if}\ \mathit{BExp}\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}$$

is equivalent to defining the operation symbol

$$\texttt{if\_then\_else\_} : \ \mathit{BExp} \times \mathit{Stmt} \times \mathit{Stmt} \to \mathit{Stmt}$$

The algebraic notation has several advantages when defining extensions or reasoning about programs is desired. For example, algebraic signatures naturally extend into algebraic specifications by adding structural identities, or equations, to an algebraic signature. This way, one can smoothly define lists (associative binary operations), sets (associative, commutative and idempotent binary operations), bags (associative and commutative binary operations), maps (sets of pairs key/value together with a few more constraints), etc., over any syntactic categories. Such structures are useful for defining both the syntax of some programming languages and especially for defining what we call in $\mathbb{K}$ the "syntax of the semantics", that is, the additional syntax needed to give semantics to the target language (configurations, auxiliary operations, etc.).

A programming language semantics is typically driven by syntax, but it often needs additional semantic data in order to properly capture the desired semantics of each language construct. Such data may include a program environment mapping program variables to memory locations, a store mapping memory locations to values, one or more stacks for functions and exceptions, a multi-set (or bag) of threads, a set of held locks associated to each thread, and so on. As seen above, in $\mathbb{K}$ such data are stored in *configurations*. To distinguish the various semantic components from each other, in $\mathbb{K}$ we "wrap" them within suggestively named *cells* when we put them together in a configuration. These cells are nothing but constructors taking the desired structure and yielding a configuration item. For example, a state cell can be defined as an operation

$$\mathsf{state} : \mathit{Map} \to \mathit{CfgItem}$$

where *Map* is the sort of maps from identifiers to integer numbers. Cells can be nested. We do not insist on how one can/should define configurations, as different implementations/realizations/encodings of $\mathbb{K}$ may choose different representations and notations. The important point is that configurations, no matter how complex, can be defined as appropriate algebraic specifications.

Therefore, all the $\mathbb{K}$ syntax and configuration declarations can be mechanically desugared into elements of rewriting logic without losing anything in the translation process, and the current $\mathbb{K}$ tool implements such a translation. The processing of the $\mathbb{K}$ rules is trickier and the current $\mathbb{K}$ tool implements a concurrency-losing translation, allowing the user also to interfere with, or configure, the translation process. Before we discuss these trade-offs of the current translation of $\mathbb{K}$ into rewriting logic, let us first formally define the notion of a $\mathbb{K}$ *rule* and the desired concurrent $\mathbb{K}$ semantics.

Given $\mathcal{W} = \{\square_1, \ldots, \square_n\}$, named *context variables*, or *holes*, a *$\mathcal{W}$-context over* $\Sigma(X)$ (assume that $X \cap \mathcal{W} = \emptyset$) is a term $k \in T_\Sigma(X \cup \mathcal{W})$ in which each variable in $\mathcal{W}$ occurs once. The instantiation of a $\mathcal{W}$-context k with an n-tuple $\bar{t} = (t_1, \ldots, t_n)$, written $k[\bar{t}]$ or $k[t_1, \ldots, t_n]$, is the term $k[t_1/\square_1, \ldots, t_n/\square_n]$. One can regard $\bar{t}$ as a substitution $\bar{t} : \mathcal{W} \to T_\Sigma(X)$, defined by $\bar{t}(\square_i) = t_i$, in which case $k[\bar{t}] = \bar{t}(k)$. In what follows we fix a signature Σ and a set of variables X.

Definition 1. *[89, 93, 91] A* $\mathbb{K}$ rule $\rho : k[\ L \to R\ ]$ *is a triple where:* k *is a* $\mathcal{W}$*-context over* $\Sigma(X)$*, called the* rule pattern*, where* $\mathcal{W}$ *are the* holes *of* k*;* k *can be thought of as the "read-only" part or the "local" context of* ρ*; and* $L, R : \mathcal{W} \to T_\Sigma(X)$ *associate to each hole in* $\mathcal{W}$ *the* original term *and its* replacement term*, resp.;* L*,* R *can be thought of as the "read/write" part of* ρ*. When* $\mathcal{W} = \{\square_1, \cdots, \square_n\}$ *and* $L(\square_i) = l_i$ *and* $R(\square_i) = r_i$*, we may write*

$$k[\ \underset{r_1}{\underline{l_1}}, \ldots, \underset{r_n}{\underline{l_n}}\ ]$$

instead of $k[\ L \to R\ ]$*, since the holes are implicit and need not be mentioned.*

The variables in $\mathcal{W}$ are only used to identify the positions in k where rewriting takes place; in practice we typically use the compact notation above, that is, underline the to-be-rewritten subterms in place and write their replacement underneath. Σ includes all the needed syntactic categories, that is, the language syntax, the configuration syntax, auxiliary operations, etc.

We can associate to any $\mathbb{K}$ rule $\rho : k[\ L \to R\ ]$ a regular rewrite rule $\mathit{K2R}(\rho) : L(k) \to R(k)$. This translation is used, for example, in our current implementation of $\mathbb{K}$ by translation to Maude. For example, the $\mathbb{K}$ rule for IMP assignment in Section 3.1 gets translated into a rewrite rule of the form:

$$\langle x \texttt{ := } i \curvearrowright \mathit{rest}\rangle_{\mathsf{k}}\ \langle \mathit{before}\ x \mapsto j\ \mathit{after}\rangle_{\mathsf{state}} \to \langle \mathit{rest}\rangle_{\mathsf{k}}\ \langle \mathit{before}\ x \mapsto i\ \mathit{after}\rangle_{\mathsf{state}}$$

Note that the ellipses " ⋯ " (representing the volatile part of the term) are now interpreted as syntactic sugar for rule variables having the appropriate collection sort (given by the type of the cell). Although the potential for concurrency with sharing of resources is reduced by this translation (as concurrent applications of rules in rewriting logic are only allowed if the rules do not overlap), it is acceptable in many cases. Conversely, given a conventional rewrite rule $\tau : \mathit{left} \to \mathit{right}$, we can generate an obvious (zero-sharing) $\mathbb{K}$ rule $\mathit{R2K}(\tau) : \square[\ \mathit{left} \to \mathit{right}\ ]$. For this reason, we take the liberty to write zero-sharing $\mathbb{K}$ rules using the conventional rewrite rule notation, as we did with several of the $\mathbb{K}$ rules in Sections 3.1 and 3.2. If τ is a rewriting logic rule, then $t \xrightarrow{\tau} t'$ denotes the binary rewrite relation generated by τ, i.e: t rewrites to t' via an instance of τ. As usual, $\xrightarrow{\tau^*}$ is the reflexive and transitive closure of $\xrightarrow{\tau}$.

The concurrent $\mathbb{K}$ rewriting relation is more complex to define than the conventional concurrent term rewriting relation. That is because we want it to be *as concurrent as possible*, so that concurrent languages or calculi defined in $\mathbb{K}$ do not just have the standard concurrent semantics of rewriting logic, which forbids overlaps between concurrent redexes, but instead have greater concurrency by allowing overlaps between redexes, provided the overlaps only happen in their read-only portions. This means that two or more concurrent rewrites can simultaneously *share* some common portion of the state. The key to achieving this is to take into account the specifics of the $\mathbb{K}$ rules, namely the fact that they are explicit about which parts are shared and which parts are rewritten. Non-conflicting $\mathbb{K}$ rules are expected to possibly be applied concurrently, like transactions do, where by "non-conflicting" rules we mean that neither of them rewrites portions of the term that are accessed (shared or written) by the other. We currently define $\mathbb{K}$'s concurrent rewrite relation in terms of *graph rewriting* (the double pushout approach), making crucial use of the notion of *parallel independence* [95]. We refer the interested reader to [90, 91] for details. What is relevant here is the fact that a $\mathbb{K}$ concurrent rewrite relation that captures the desired rules-as-transactions informal semantics discussed above *can* be defined; we denote it $\Rrightarrow$ instead of $\rightarrow$. While rewriting logic can theoretically capture the intended concurrent semantics of graph rewriting [96], the representation in [96] is impractical. For that reason, in our implementation of the $\mathbb{K}$ tool we currently follow a different path, as explained below.

Let us exemplify $\Rrightarrow$ on the $\mathbb{K}$ semantics of IMP and IMP++. Since in $\mathbb{K}$ rule instances can share read-only data, various (actually all matching) instances of the lookup rule can apply concurrently, in spite of the fact that they overlap on the state subterm. Similarly, since the rules for variable assignment and increment declare volatile everything else in the state except the mapping corresponding to the variable, multiple assignments and increments of distinct variables can happen concurrently. However, if two threads want to write the same variable, or if one wants to write it while another wants to read it, then the two corresponding rules need to be interleaved, because the two rule instances are in a concurrency conflict. Note also that the rule for `print` matches and changes the end of the output cell; that means, in particular, that multiple outputs by various threads need to be interleaved for the same reason as above. On the other hand, the rule for `spawn` matches any empty top-level position and replaces it by the new thread, so threads can spawn other threads concurrently. Similarly, multiple threads can be dissolved concurrently when they are done. These concurrency aspects of IMP++ can be defined formally thanks to the specific nature of the $\mathbb{K}$ rules. If instead we had used standard rewrite rules instead of $\mathbb{K}$ rules, many of the structure-sharing concurrent steps above would need to be interleaved, because rewrite rule instances which overlap cannot be applied concurrently.

$\mathbb{K}$'s rewriting has the following properties, where $t \overset{\rho_1+\cdots+\rho_n}{\Rrightarrow} t'$ means that t can be rewritten in *one concurrent step* to t' using rules $\rho_1, \ldots, \rho_n$:

Theorem 1. *[93, 91] Let ρ, $\rho_1, \ldots, \rho_n$ be not necessarily distinct $\mathbb{K}$ rules.*

<u>Completeness:</u> *If $t \xrightarrow{K2R(\rho)} t'$ then $t \overset{\rho}{\Rrightarrow} t'$.*

<u>Soundness:</u> *If $t \overset{\rho}{\Rrightarrow} t'$ then $t \xrightarrow{K2R(\rho)} t'$.*

Serializability: *If* $t \overset{\rho_1+\cdots+\rho_n}{\Rrightarrow} t'$*, then there exists a sequence of terms* $t_0, \cdots, t_n$*, such that* $t_0 = t$*,* $t_n = t'$*, and* $t_{i-1} \overset{\rho_i}{\Rrightarrow} t_i$*.*

Completeness says that any steps made using rewriting logic can also be made using $\mathbb{K}$ rewriting. Soundness states that any non-concurrent step made using $\mathbb{K}$ rewriting corresponds to zero, one or more rewriting logic steps; this is due to the fact that the term to be rewritten is represented as a graph in $\mathbb{K}$, and zero, one or more term-rewrite steps are needed to mimic a graph rewrite step (zero when the rewritten part is unreachable). The serializability result says that the concurrent rewrite relation $\Rrightarrow$ does not reach any other terms than the concurrent rewrite relation $\to$: it just reaches them in a possibly smaller number of steps.

From a practical viewpoint, the theorem above tells us that it may be acceptable, in many situations, to translate $\mathbb{K}$ rules into conventional rewrite rules using the *K2R* map. The only thing lost in translation is the amount of true concurrency available in the original $\mathbb{K}$ definition. Note, however, that most semantic frameworks for programming languages follow an interleaving philosophy by their nature, so "losing some true concurrency" cannot even be formulated in those frameworks. Nevertheless, we believe that with the advance of massively parallel architectures, maximizing the true concurrency capability of a semantic framework will be increasingly desirable, so $\mathbb{K}$ makes no compromises in what regards its theoretical support for concurrency. That being said, the reader who thinks that $\mathbb{K}$'s concurrent rewrite relation $\Rrightarrow$ is hard to realize, or who does not want to get into the technicalities of graph rewriting, or who simply does not believe in true concurrency, is free to replace it in the rest of this section with the (still truly concurrent but not structure-sharing) rewriting logic relation $\to$ associated to it via *K2R*. The remainder of this section is parametric in the relation $\Rrightarrow$.

Definition 2. *A* $\mathbb{K}$ (rewrite) system *(or* $\mathbb{K}$ theory *or* $\mathbb{K}$ definition*) is a triple* $\mathcal{K} = (\Sigma, \mathcal{S}, \mathcal{C})$*, where* Σ *is its* signature *and* $\mathcal{S}$ *and* $\mathcal{C}$ *are sets of* structural *and* computational $\mathbb{K}$ *rules, respectively. Let* $\Rrightarrow_{\mathcal{S}}$ *and* $\Rrightarrow_{\mathcal{C}}$ *be the corresponding concurrent rewrite relations, and let* $\Rrightarrow_{\mathcal{K}}$ *be the relation* $\Rrightarrow_{\mathcal{S}}^{*} \circ \Rrightarrow_{\mathcal{C}} \circ \Rrightarrow_{\mathcal{S}}^{*}$*.*

In short, a concurrent rewrite step in a $\mathbb{K}$ definition can be thought of as a (concurrent) computational step *modulo* structural rearrangements. From a rewriting logic perspective, the structural rewrite rules in $\mathcal{S}$ can be thought of as "half-equations", in the sense that they have the same intuition as rewriting logic's equations (namely that of non-computational rearrangements of the term to rewrite), but they are oriented left-to-right. Although operationally speaking a structural rule fulfills only half the job of an equation, we can always obtain the same effect of an equation by providing an additional inverse structural rule, from right to left, as we do, for example, with the heating/cooling rules corresponding to evaluation strategies (see Section 3.1). In fact, bi-directional structural rules are heavily used to define $\mathbb{K}$ configurations, as configurations typically contain lists, sets, multi-sets, maps, etc., and these data types are best defined using equations (such as associativity, commutativity, etc.). Implementations of $\mathbb{K}$, like implementations of rewriting logic, will likely provide special builtin support for certain bi-directional structural rules such as associativity, commutativity, etc.; for example, our current $\mathbb{K}$ tool translates those into Maude operator attributes or equations and then relies on Maude's builtin and efficient support for those.

Therefore, $\Rrightarrow_{\mathcal{S}}$ is not necessarily symmetric. Moreover, note that $t \Rrightarrow^{*}_{\mathcal{S}} u$ and $t \Rrightarrow_{\mathcal{K}} t'$ and $u \Rrightarrow_{\mathcal{K}} u'$ do not necessarily imply $t' \Rrightarrow^{*}_{\mathcal{S}} u'$. To see why this makes practical sense, consider a hypothetical programming language which already provides a statement **`halt`** for abrupt termination whose semantics is given with a computational rule (dissolving the entire contents of the `k` cell) and suppose that we want to add a non-deterministic halting statement, say **`ndhalt`**. One way to do it is to add a structural rule rewriting **`ndhalt`** to **`halt`** and a computational rule dissolving the **`ndhalt`** statement (as if it was the empty statement). Then take t to be some configuration *cfg*[**`ndhalt;rest`**], u to be *cfg*[**`halt;rest`**], t' to be *cfg*[**`rest`**], and u' to be *cfg*[] (i.e., *cfg* with an empty computation cell). Similarly, $t \Rrightarrow^{*}_{\mathcal{S}} u$ and $t \Rrightarrow_{\mathcal{K}} t'$ and $t' \Rrightarrow^{*}_{\mathcal{S}} u'$ do not necessarily imply $u \Rrightarrow_{\mathcal{K}} u'$. For example, take the same t, u and t' as above, but $u' = t'$.

The rewrite relation $\Rrightarrow_{\mathcal{K}}$ associated to a $\mathbb{K}$ rewrite system $\mathcal{K} = (\Sigma, \mathcal{S}, \mathcal{C})$ gives us an obvious transition system on the set of ground Σ-terms T_{Σ}, which can be regarded as the semantics of $\mathcal{K}$. Thus, the semantics of $\mathbb{K}$ is given in terms of transition systems, based on a concurrent rewrite relation that takes the specific nature (e.g., explicit sharing) of the $\mathbb{K}$ rules into account. If one forgets the specific nature of the $\mathbb{K}$ rules then one still gets a valid concurrent semantics, amenable for execution on existing rewrite engines like Maude, but one which loses some of the true concurrency of the original $\mathbb{K}$ definition. $\mathbb{K}$ tools can implement different techniques and algorithms that work with $\mathbb{K}$ definitions. For example, thanks to excellent support from the underlying Maude system, our current implementation provides support for execution, for state-space search, and for explicit-state LTL model-checking. While the current implementation of the $\mathbb{K}$ tool heavily relies on Maude, term rewriting using $\mathbb{K}$ rules can be theoretically implemented more efficiently than Maude, because $\mathbb{K}$ requires less support than Maude offers; for example, $\mathbb{K}$ does not require conditional rewrite rules in their full generality. This hypothesis will be tested soon, since a prototype $\mathbb{K}$ rewrite engine is under development (check $\mathbb{K}$'s website for news and progress).

Since the $\mathbb{K}$ tool is being used to define real and complex programming languages, such as C (see Section 4), performance is a crucial aspect which has been given a high priority in the design of the tool in general and in the translation from $\mathbb{K}$ to rewriting logic in particular. For example, to avoid the non-termination given by the bi-directional structural rules corresponding to evaluation strategies, the tool applies them by default from-left-to-right when the hot expression (i.e., the expression on top of the computation structure) can still be evaluated, and from-right-to-left when the hot expression is a result. Moreover, all structural rules are by default translated into Maude equations, rather than rewrite rules. These default choices lose some non-determinism and may even be logically incorrect at the theoretical level, but avoid non-termination, significantly increase performance, and are correct in practice when executed (because Maude always applies the equations from-left-to-right). To give users freedom in tuning up the tool for their needs, the current implementation actually allows the users to interfere with the translation process by means of configurable translation options. For example, the use of the $\mathbb{K}$ tool can explicitly state which operations are desired to yield full non-deterministic evaluation strategies (but then one needs to use search) and can say which $\mathbb{K}$ rules should be translated into Maude equations rather than rules. Of course, the correctness of the translation is then the user's responsibility. The $\mathbb{K}$ tool is available for download and online experimentation at `http://k-framework.org`.

4. Programming Language Semantics

Having formal semantics for real programming languages, regardless of the formalism that is being used, is undoubtedly a very important step, useful not only to help us understand those languages better but also to serve as a solid foundation for implementations and for program analysis and verification techniques and tools. Using rewriting logic as a formalism for such semantics has the additional benefit that such techniques and tools can be *directly derived* from the language semantics with minimal effort, as shown throughout this paper.

The rewriting logic semantics technique described in Section 3 has been used to define several programming languages or large fragments of them. Some of these languages serve as models for teaching various language paradigms, which we do not mention here but can be found on webpages for programming language courses at UIUC and can be reached from `http://k-framework.org`, while others are real programming languages, such as C [97], Scheme [98], or Java 1.4 [46, 45]. In this section we only briefly discuss the rewrite logic semantics of C [97], more precisely of the ISO/IEC 9899:1999 (C99) standard, as formalized by Chucky Ellison using the $\mathbb{K}$ framework. This semantics is currently being used by several researchers and research groups, both directly in their tools and indirectly as a basis for understanding (and sometimes criticizing) the C language. This has led to the "C Semantics" Google code project repository at `http://c-semantics.googlecode.com/`.

The $\mathbb{K}$ semantics of C defines approximately 150 C syntactic operators and many other intermediate or auxiliary semantic operators. The definitions of these operators are given by 1,163 semantic rules spread over 5884 lines of $\mathbb{K}$ code (LOC). However, it takes only 77 of those rules (536 LOC) to cover the behavior of statements, and another 163 for expressions (748 LOC). There are 505 rules for dealing with types, 115 rules for memory, and 189 rules defining other necessary mechanisms. Finally, there are 114 rules for the core of our standard library.

This is the most comprehensive formal semantics of C to date. Figure 3 shows a summary, in terms of features defined and how completely they were

Feature	GH	CCR	CR	No	Pa	BL	ER
Bitfields	●	◐	○	○	◐	○	●
Enums	◐	●	○	○	●	○	●
Floats	○	○	○	○	●	●	●
String Literal	○	●	○	○	●	●	●
Struct as Value	○	○	○	●	○	○	●
Arithmetic	◐	●	●	○	●	●	●
Bitwise	○	●	○	○	●	●	●
Casts	◐	◐	○	◐	◐	●	●
Functions	●	●	◐	●	●	●	●
Exp. Side Effects	●	●	○	●	●	○	●
Break/Continue	◐	●	◐	●	●	●	●
Goto	◐	○	○	○	●	○	●
Switch	◐	●	○	○	●	◐	●
Longjmp	○	○	○	○	○	○	●
Malloc	○	○	○	○	○	○	●
Variadic Funcs.	○	○	○	○	○	○	●

●: Fully Described

◐: Partially Described

○: Not Described

GH represents Gurevich and Huggins [99], *CCR* is Cook et al. [100], *CR* is Cook and Subramanian [101], *No* is Norrish [102], *Pa* is Papaspyrou [103], *BL* is Blazy and Leroy [104], and *ER* is our work, Ellison and Roşu [97].

Figure 3: Comparison of the most comprehensive C semantics to date

defined, of some of the most comprehensive C semantics available. Our semantics is executable and has been thoroughly tested. All aspects related to the features mentioned below are given a direct semantics. *Expressions*: referencing and dereferencing, casts, array indexing, structure members, arithmetic, bitwise, and logical operators, sizeof, increment and decrement, assignments, sequencing, ternary conditional; *Statements*: for, do-while, while, if/else, switch, goto, break, continue, return; *Types and Declarations*: enums, structs, unions, bitfields, initializers, typedefs; *Values*: regular scalar values (signed/unsigned arithmetic and pointer types), structs, unions; *Standard Library*: malloc/free, set/longjmp, basic I/O; *Environment*: command line arguments; *Conversions*: (implicit) argument and parameter promotions and arithmetic conversion, and (explicit) casts.

No matter what the intended use is for a formal semantics, such a use is limited if one cannot achieve confidence in its correctness. To achieve this aim, executable semantics has an immense practical advantage over non-executable semantics, because one can simply test it. The C semantics in [97] has been encapsulated inside a drop-in replacement for Gnu's C Compiler (GCC), called "KCC". This allows one to test the semantics as one would test a compiler:

```
$ kcc helloworld.c
$ ./a.out
Hello world
```

Indeed, the C semantics has been successfully run against all the examples in the Kernigham and Ritchie manual that supposedly cover all the features of ANSI C. Moreover, a series of challenging C programs collected from the Internet, such as programs from the Obfuscated C programming competition, totaling more than 10,000 LOC are included in the regression tests of the C semantics, so these are all executed each time the semantics is changed. In addition to the above, the GCC C-torture-test (which contains 776 C programs conforming to the standard semantics of C99) has been executed in the C semantics and its behavior compared to that of GCC itself, as well as to Intel's C Compiler (ICC) and to the LLVM C compiler, Clang.

C is so complex that even dedicated and broadly used compilers like GCC or ICC cannot compile and execute all the programs in the GCC torture-test. All in all, considering all the tests that the C semantics has been tested on, the GCC compiler successfully passed 99% of them (768 tests), ICC passed 99.4% (761 tests), Clang passed 98.3% (763 tests), while our C semantics (compiled into Maude using the $\mathbb{K}$ tool) passed 99.2% of them (770 tests). The C semantics ran over 90% of these programs in under 5 seconds (each). An additional 6% completed in 10 minutes, 1% in 40 minutes, and 2% further in under 2 days. The remaining programs either did not finish because they were computationally very intensive, or they made use of combinations of features whose semantics is not clear in the ISO/IEC 9899:1999 (C99). While this is not terribly fast performance, especially when compared to compiled C, the reader should keep in mind that this is an interpreter obtained *for free* from a formal semantics and that other existing semantics of C are either "paper" definitions (e.g., [99]), or not executable (e.g., [102]), or very slow (e.g., we were not able to execute factorial of 6 or the 4th Fibonacci's number using the Haskell-based definition in [105, 103]), or covering only a C fragment (e.g., [104]). Moreover, our semantics of C can be used *directly* and *unchanged* for other purposes, such as for model checking (see Section 9) and for deductive verification (see Section 10).

As the comparison above of our C semantics with existing compilers suggests, the user of our C executable semantics in fact cannot distinguish it from an actual implementation of C, except for the execution speed, and this was intentional. Indeed, there is no reason why an executable specification should behave any different in terms of executability from an implementation. While lower execution performance can be seen as the price to pay for language specifications being mathematically grounded and thus amenable for formal reasoning, we strongly believe that even the execution speeds of language specifications can be significantly improved with the implementation of specialized $\mathbb{K}$ rewrite engines (which is ongoing work).

5. Real-Time Language Semantics

Three real-time programming languages have been given formal semantics as real-time rewrite theories [71] in Real-Time Maude [72]. Using the model checking features of Real-Time Maude it then becomes possible to formally analyze programs in such languages.

In [106], AlTurki et al. present a language for real-time concurrent programming for industrial use in DOCOMO Labs called L. The goal of L is to serve as a programming model for higher-level software specifications in SDL or UML. A related goal is to support formal analysis of L programs by both real-time model checking and static analysis, so that software design errors can be caught at design time. The way all this is accomplished is by giving a formal semantics to L in Real-Time Maude, which automatically provides an interpreter and a real-time model checker for L. Static analysis capabilities are added to L by using Maude to define an *abstract semantics* for L in rewriting logic, which is then used as the static analyzer.

The Orc model of real-time concurrent computation [107, 108, 109] has been given semantics in rewriting logic using real-time rewrite theories [63, 110]. Although Orc is a very simple and elegant language, its real-time semantics is quite subtle for two reasons. First, in the evaluation of any Orc expression, internal computation always has higher priority than the handling of external events; this means that, even without modeling time, a vanilla-flavored SOS semantics is not expressive enough to capture these different priorities: two SOS relations are needed [108]. Second, Orc is by design a real-time language, where time is a crucial feature. Using real-time rewrite theories, this double subtlety of the Orc semantics was faithfully captured in [63]; furthermore, this semantics yielded of course an Orc interpreter and a real-time model checker. But Orc is not just a model of computation: it is also a concurrent programming language. This suggested the following challenge question: can a correct-by-construction distributed Orc implementation be derived from its rewriting logic semantics? This question was answered in two stages. Since, as discussed in Section 2.5, a small-step SOS semantics is typically horribly inefficient and it was certainly so in the case of Orc, a much more efficient *reduction semantics* was first defined in [110], and was proved to be bisimilar to the small-step SOS semantics. This semantics provided a much more efficient interpreter and model checker. Furthermore, to explicitly model different Orc clients and various web sites, and their message passing communication, the Orc semantics was seamlessly extended in [110] to a distributed object-based Orc semantics, which modeled what a distributed implementation should look like. The only remaining step was to pass from this model of a distributed implementation to an actual Maude-based distributed real-time implementation. This was accomplished in [111] using three main ideas: (i) the use of sockets in Maude to actually deploy a distributed implementation; (ii) the systematic replacement of logical time by physical time, supported by Ticker objects external to Maude, while retaining the rewriting semantics throughout; and (iii) the experimental estimation of the physical time required for "zero-time" Maude subcomputations, to ensure that the granularity of time ticks is such that all "instantaneous transitions" have already happened before the next tick.

Creol is an object-oriented language supporting concurrent objects which communicate through asynchronous method calls. Its rewriting-logic-based operational semantics was defined in [47] without real-time features. However, to support applications such as sensor systems with wireless communication, where messages expire and may collide with each other, Creol's design and operational semantics have been extended in [112] to Timed Creol using rewriting logic. The notion of time used by Timed Creol is described as a "lightweight" one in [112]. Time is discrete and is represented by a time object. This approach does not require a full use of the features in Real-Time Maude (Maude itself is sufficient to define the real-time semantics). The effectiveness of Timed Creol in the modeling and analysis of applications such as sensor networks is illustrated in [112] through a case study. The timed semantics of Creol in [112] has been extended with deadlines and user-defined schedules in the ABS language [113].

6. Semantics of Modeling Language

Modeling languages are quite useful, but they can be made even more useful by substantially increasing their analytic power through formal analysis, since this can make it possible to catch expensive design errors very early. Formal analysis is impossible or fraudulent without a formal semantics. Early work in developing rewriting-logic-based formal semantics focused on object-oriented design notations and languages [114, 115, 116], and stimulated subsequent work on UML and UML-like notations, e.g., [117, 118, 119, 120, 121, 122, 123, 124, 125].

A more ambitious question is: can we give semantics not just to a single modeling language, but to an entire *modeling framework* where different modeling languages can be defined? This question has been answered positively in [126, 127, 128, 129, 130]. This line of research has led to MOMENT2, an algebraic model management framework and tool written in Maude and developed by Artur Boronat [127]. It permits manipulating software models in the Eclipse Modeling Framework (EMF). It uses OMG standards, such as Meta-Object Facility (MOF), Object Constraint Language (OCL) and Query/View/Transformation (QVT), as a clean interface between rewriting-logic-based formal methods and model-based industrial tools. Specifically, it supports formal analyses based on rewriting logic and graph transformations to endow model-driven software engineering with strong analytic capabilities. MOMENT2 supports not just one fixed modeling language, but any modeling language whose *meta-model* is specified in MOF. In more detail, a modeling language is specified as a pair $(\mathcal{M}, \mathcal{C})$, where $\mathcal{M}$ is its MOF-based metamodel, and $\mathcal{C}$ are the OCL constraints that $\mathcal{M}$ should satisfy. Using rewriting-logic-based reflection and its efficient support in Maude, MOMENT2 provides an *executable algebraic semantics* for such metamodel specifications $(\mathcal{M}, \mathcal{C})$ in the form of a theory $\mathbb{A}(\mathcal{M}, \mathcal{C})$ in membership equational logic (MEL) [131], so that a model M conformant with the metamodel $(\mathcal{M}, \mathcal{C})$ is exactly a term of sort *Model* in $\mathbb{A}(\mathcal{M}, \mathcal{C})$, and so that satisfaction of OCL constraints is also decidable using the algebraic semantics [132, 130].

Due to the executability of MEL specifications in Maude, the realization of MOF metamodels as MEL theories enhances the formalization and prototyping of model-driven development processes, such as: (i) model transformations; (ii) model-driven roundtrip engineering; (iii) model traceability; and (iv) model management. These processes permit, for example, merging models, generating mappings between models, and computing differences between models; they can be used to solve complex scenarios such as the roundtrip problem. In MOMENT2 the formal semantics of *model transformations* is given by rewrite theories specified in a user-friendly QVT-based syntax [128]. Such model transformations can describe the dynamic evolution of systems at the level of their models. Using the search and LTL model checking features of Maude, properties about the dynamic evolution of a model M conformant with a metamodel specification $(\mathcal{M}, \mathcal{C})$ can then be formally analyzed by model checking [128]. Real-time modeling languages can likewise be supported and analyzed [133]; this is further discussed below.

6.1. Semantics of Real-Time Modeling Languages

There is strong interest in modeling languages for real-time and embedded systems. The rewriting logic semantics for such modeling languages can be naturally based on real-time rewrite theories. Using a tool like Real-Time Maude, what this means in practice is that such models can then be simulated; and that their formal properties, in particular their safety requirements, can be model checked. Furthermore, the simulations and formal analysis capabilities added to the given modeling language can be offered as “plugins” to already existing modeling tools, so that much of the formal analysis happens “under the hood,” and somebody already familiar with the given modeling notation can perform such formal analysis without having an in-depth understanding of the underlying formalism.

The Ptolemy II modeling language [81] supports design and simulation of concurrent, real-time, embedded systems expressed in several models of computation (MoCs), such as state machines, data flow, and discrete-event models, that govern the interaction between concurrent components. A user can visually design and simulate hierarchical models, which may combine different MoCs. Furthermore, Ptolemy II has code generation capabilities to translate models into other modeling or programming languages such as C or Java. Discrete-Event (DE) Models are among the most central in Ptolemy II. Their semantics is defined by the *tagged signal model* [134]. The work by Bae et al. in [135] endows DE models in Ptolemy II with formal analysis capabilities by: (i) defining a semantics for them as real-time rewrite theories; (ii) automating such a formal semantics as a model transformation using Ptolemy II's code generation features; (iii) providing a Real-Time Maude plugin, so that Ptolemy II users can use an extended GUI to define temporal logic properties of their models in an intutitive syntax and can invoke Real-Time Maude from the GUI to model check their models. This work has been further advanced in [136] to support not just flat DE models, but *hierarchical* ones. That is, above tasks (i)–(iii) have been extended to hierarchical DE models; this extension is nontrivial, because it requires combining synchronous fixpoint computations with hierarchical structure.

AADL (`http://www.aadl.info/`) is a standard for modeling embedded systems that is widely used in avionics and other safety-critical applications. However, AADL lacks a formal semantics, which severely limits both unambiguous communication among model developers and the formal analysis of AADL models. In [137] Ölveczky et al. define a formal object-based real-time concurrent semantics for a behavioral subset of AADL in rewriting logic, which includes the essential aspects of AADL's behavior annex. Such a semantics is directly executable in Real-Time Maude and provides an AADL simulator and LTL model checking tool called *AADL2Maude*. *AADL2Maude* is integrated with OSATE, so that OSATE's code generation facility is used to automatically transform AADL models into their corresponding Real-Time Maude specifications. Such transformed models can then be executed and model checked by Real-Time Maude. One difficulty with AADL models is that, by being made up of various hierarchical components that communicate asynchronously with each other, their model checking formal analysis can easily experience a combinatorial explosion. However, many such models express designs of distributed embedded systems which, while being asynchronous, should behave in a virtually synchronous way. This suggest the possibility of using the PALS pattern [138], which reduces distributed real-time systems with virtual synchrony to synchronous ones, to pass from simple synchronous systems, which have much smaller state spaces and are much easier to model check, to semantically equivalent asynchronous systems, which often cannot be directly model checked but can be verified indirectly through their synchronous counterparts. This has led to the design of the Synchronous AADL sublanguage in [139], where the user can specify synchronous AADL models by using a sublanguage of AADL with some special keywords. A synchronous rewriting semantics for such models has also been defined in [139]. Using OSATE's code generation facility, synchronous AADL models can be transformed into their corresponding Real-Time Maude specifications in the *SynchAADL2Maude* tool, which is provided as a plugin to OSATE. Likewise, the user can define temporal logic properties of synchronous AADL models based on their features, without requiring knowledge of the underlying formalism, and can model check such models in Real-Time Maude.

A more ambitious goal is to provide a *framework*, where a wide range of real-time Domain-Specific Visual Languages (DSVLs), as well as their dynamic real-time behavior, can be specified with a rigorous semantics. This is precisely the goal of two frameworks and associated tools: (i) the *e-Motions* framework [140]; and (ii) *MOMENT2*'s support for real-time DSVLs [133].

- In *e-Motions*, DSVLs are specified by their corresponding metamodels, and dynamic behavior is specified by rules that define in-place model transformations. But the goals of *e-Motions* do not remain at the syntax/visual level: they also include giving a precise rewriting logic semantics in Real-Time Maude to the different real-time DSVLs that can be defined in *e-Motions*, and to automatically support simulation and formal analysis of models by using the underlying Real-Time Maude engine. The formal semantics translates the metamodel of a DSVL as an object class, the corresponding models as object configurations of that class, and the *e-Motions* rules as rewrite rules. Since all these translations are automatic and define a DSVL's formal semantics, a modeling language designer using *e-Motions* does not have to explicitly define the DSVL's formal semantics: it comes for free, together with the simulation and model checking features, once the DSVL's metamodel and the dynamic behavior rules are specified.

- In [133], the *MOMENT2* framework has been extended to support the formal specification and analysis of real-time model-based systems. This is achieved by means of a collection of built-in timed constructs for defining the timed behavior of such systems. Timed behavior is specified using in-place model transformations. Furthermore, the formal semantics of a *timed behavioral specification* in *MOMENT2* is given by a corresponding real-time rewrite theory. In this way, models can be simulated and model checked using MOMENT2's Maude-based analysis tools. In addition, by using in-place multi-domain model transformations in *MOMENT2*, an existing model-based system can be extended with timed features in a non-intrusive way, in the sense that no modification is needed for the class diagram.

6.2. Semantics of Ptolemy and AADL through Examples

To give a feeling for what the rewriting logic semantics of real-time modeling languages looks like, we illustrate with examples two such modeling languages, namely, Ptolemy II and (Synchronous) AADL, and give a high-level summary of their respective rewriting logic semantics. Due to space limitations, many details are omitted. We refer to the longer studies [141], [142], and [143], from which the examples and semantic definitions below are adapted, for more details. In particular, the recent [143] gives a very good overview of the uses of Real-Time Maude in giving formal semantics to, and providing formal analysis for, various real-time modeling languages.

6.2.1. A Ptolemy II Example and its Semantics.

As already mentioned, Ptolemy II [81] is a widely used graphical modeling and simulation tool for real-time and embedded systems. In Ptolemy II, real-time systems are modeled as *discrete-event* (DE) models, which consist of a set of components called *actors*, having *input ports* and *output ports*, and linked by communication channels that pass *events* from one port to another. Such a model can be encapsulated as a *composite* actor, which may also have input and output ports. Each event has two components: a *tag* and a *value.* A tag t is a pair (τ, n), with τ a positive real called the *timestamp*, and n a natural number called the *microstep index.*

In each iteration of the system, all components with input execute *synchronously.* That is, since connections are instantaneous and the components execute in lock-step, we must compute the *fixpoint* of the input for each component in the round before its execution; this input comes from the output of another component's execution in the same synchronous round.

Figure 4 shows a hierarchical Ptolemy II model of a fault-tolerant traffic light system at a pedestrian crossing, consisting of one car light and one pedestrian light. Each light is represented by a set of *set variable* actors (`Pred` and `Pgrn` represent the pedestrian light, and `Cred`, `Cyel`, and `Cgrn` represent the car light). A light is *on* iff the corresponding variable has the value 1. The Finite State Machine (FSM) actor `Decision` "generates" failures and repairs by alternating between staying in location `Normal` for 15 time units and staying in location for `Abnormal` for 5 time units, and by sending events to the `TrafficLight` through its `Error` and `Ok` ports accordingly. During `normal` operations, the lights are controlled by the FSM actors `CarLight` and `PedestrianLight` (their FSM's are not shown in the figure) that send values to set the variables; in addition, `CarLight` sends signals to the `PedestrianLight` actor through its `Pgo` and `Pstop` output ports.

We now summarize the rewriting logic semantics of Ptolemy II DE models in Real-Time Maude. Some details are omitted, for which we refer the reader to [141]. The semantics is defined in an object-oriented style, where the global state has the form of a *multi-set* of the form:

`{`*actors connections* `< global : EventQueue | queue :` *event queue* `>}`

where *actors* are objects modeling the actor instances in the Ptolemy model, *connections* are its connections, and *event queue* denotes the global event queue.

Each Ptolemy II actor is modeled as an object instance of a subclass of the class `Actor`, that contains the *ports* and the *parameters* of the actor. Composite actors add an attribute, `innerActors`, denoting its inner actor objects and connections:

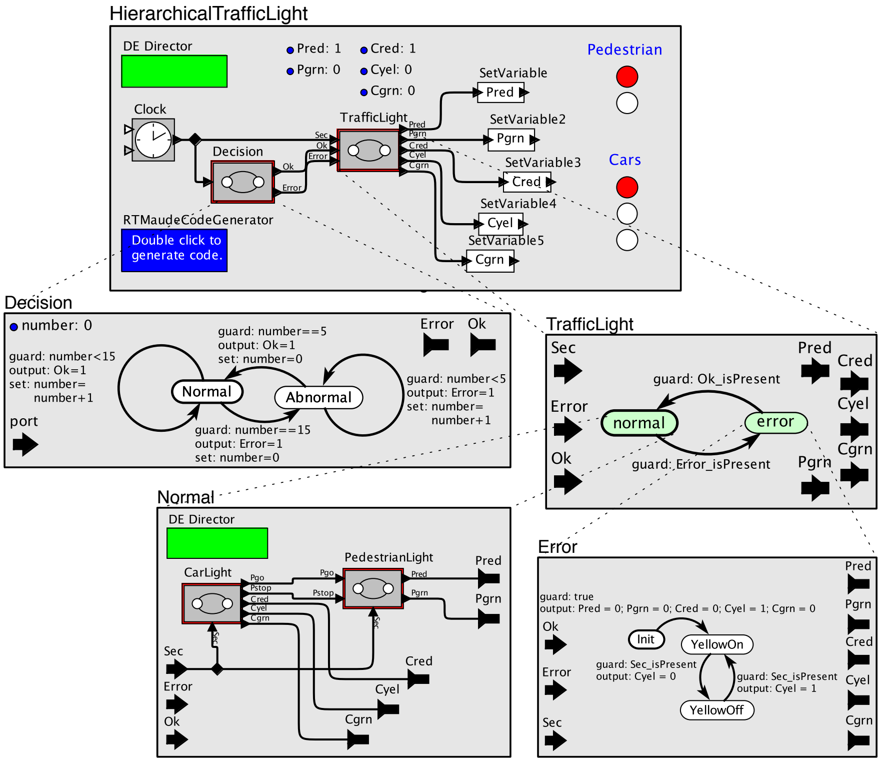


Figure 4: A hierarchical fault-tolerant traffic light system in Ptolemy II.

```
class Actor | ports : Configuration, parameters : Configuration .
class CompositeActor | innerActors : Configuration .
class AtomicActor .
subclass CompositeActor AtomicActor < Actor .
```

A *port* is represented as an object with a name (the identifier of the port object), a status (`unknown`, `present`, or `absent`, denoting the “current” knowledge about whether there is input/output in the current iteration), and a `value`:

```
class Port | status : PortStatus, value : Value .
class InPort .    class OutPort .     subclass InPort OutPort < Port .
sort PortStatus .
ops unknown present absent : -> PortStatus [ctor] .
```

The semantics has three rewrite rules and several equations used to compute fixpoints. The first rule is a ‘tick’ rule that advances time until the first events in the event queue are scheduled (and reduces the remaining time of the other events according to the elapsed time).

```
vars SYSTEM : ObjectConfiguration .  var EVTS : Events .
var QUEUE : EventQueue .  var NZT : NzTime .  var N : Nat .

rl [tick] :
   {SYSTEM  < global : EventQueue | queue : (EVTS ; NZT ; N) :: QUEUE >}
  =>
   {delta(SYSTEM, NZT)
    < global : EventQueue | queue : (EVTS ; 0 ; N) :: delta(QUEUE, NZT) >}
  in time NZT .
```

The second rule (not shown) is a "microstep tick rule" that advances "time" with some microsteps if needed to enable the first event in the event queue. The third rewrite rule below performs a synchronous step of the system when the remaining timer and microstep of the first events in the event queue are zero:

```
rl [executeStep] :
   {SYSTEM  < global : EventQueue | queue : (EVTS ; 0 ; 0) :: QUEUE >}
  =>
   {< global : EventQueue | queue : QUEUE >
    postfire(portFixPoints(releaseEvt(EVTS) clearPorts(SYSTEM)))} .
```

The function `clearPorts` sets the `status` of each port to `unknown`. The function `releaseEvt` takes all the ripe events and puts them into the corresponding output ports. The function `portFixPoints` (whose equations are not shown) computes all the port values in this round.

Since the rewriting logic semantics of Ptolemy II is *executable*, it defines an interpreter for Ptolemy II DE models, which has been integrated with the Ptolemy II tool. Although Ptolemy II models are already executable, this can be used to test the Ptolemy II implementation against the formal semantics, and also for certain forms of symbolic execution. However, the main value added to Ptolemy II is that, as shown in Section 9.2 for the traffic system example, one can invoke from Ptolemy II the Real-Time Maude model checker to verify temporal logic properties of DE models.

6.2.2. A Synchronous AADL Example and its Semantics.

As already mentioned, the *Synchronous AADL* modeling language [142] extends a subset of AADL to support the specification of real-time models that are assumed to be synchronous, at least at a high level of abstraction. This greatly increases the chances of formally analyzing such systems, while leaving open the possibility of refining such models into distributed, asynchronous ones using the PALS formal pattern [144, 138]. We can exemplify *Synchronous AADL* with fragments of a model of an avionics system based on a specification by Steve Miller and Darren Cofer at Rockwell-Collins [144]. A full description of this model is given in [145]; here we just give an impressionistic description of it and refer to [145] for additional details. The details, as such, are not the point of the example: the key point is to illustrate the idea that a synchronous AADL model can be viewed as a synchronous composition of state machines (one such machine per AADL component), which are formalized in the Real-Time Maude semantics as separate objects that change their state synchronously.

In *integrated modular avionics* (IMA), a cabinet is a chassis with a power supply, internal bus, and general purpose computing, I/O, and memory cards. Aircraft applications are implemented using the resources in the cabinets. There are always two or more physically separated cabinets on the aircraft so that physical damage does not take out the computer system. The *active standby* system considers the case of two cabinets and focuses on the logic for deciding which side is *active*. Each side can fail, and a failed side can recover after failure. In case one side fails, the non-failed side should be the active side. In addition, the pilot can toggle the active status of the sides. The full functionality of each side depends on the two sides' perception of the availability of other system components. An AADL-like graphical description of the system is shown in Figure 5, and the following is a fragment of its top-level textual representation, which declares the architecture of the system, with the three subcomponents `sideOne`, `sideTwo`, and `env`, and with immediate data connections (denoted by the arrow '`->`') from the environment to the two sides, and with delayed data connections ('`->>`') between the two sides. Each subcomponent contains a thread specification (not shown) in AADL's behavior annex.

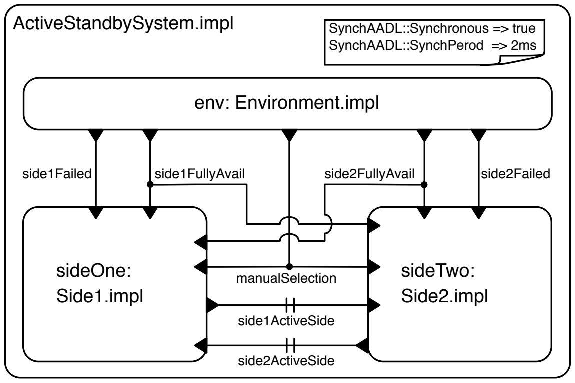


Figure 5: The architecture of the active standby system.

```
system implementation ActiveStandbySystem.impl
  properties
    SynchAADL::Synchronous => true;     SynchAADL::SynchPeriod  => 2 ms;
  subcomponents
    sideOne: system Side1.impl;   sideTwo: system Side2.impl;   env: system Environment.impl;
  connections
    data port sideOne.side1ActiveSide ->> sideTwo.side1ActiveSide;
    data port sideTwo.side2ActiveSide ->> sideOne.side2ActiveSide;
    data port env.side1FullyAvail -> sideOne.side1FullyAvail;
    data port env.side1FullyAvail -> sideTwo.side1FullyAvail;
    ...
end ActiveStandbySystem.impl;
```

We now summarize the semantics of *Synchronous AADL* in rewriting logic. The semantics of a component-based language can naturally be defined in an object-oriented style, where each component instance is modeled as an object. The hierarchical structure of *Synchronous AADL* components is reflected in the nested structure of objects, in which an attribute of an object contains its subcomponents as a multiset of objects. Any *Synchronous AADL* component instance is represented as an object instance of a subclass of the following class Component, which contains the attributes common to all kinds of components:

```
class Component | features : Configuration,   subcomponents : Configuration,
                  properties : Properties,     connections : ConnectionSet .
```

The attribute features denotes the ports of a component, represented as a multi-set of Port objects; subcomponents denotes the subcomponents of the object; properties denotes its *properties*; and connections denotes its connections.

The Thread class is declared as follows:

```
class Thread | behaviorRef : ComponentRef,  variables : Valuation,
               currState : Location,        completeStates : LocationSet .
subclass Thread < Component .
```

Given a *Synchronous AADL* system, a synchronous transition step of the system is then formalized by the following 'tick' rewrite rule:

```
var SYSTEM : Object .    var VAL : Valuation .    var VALS : ValuationSet .

crl [syncStepWithTime] :
    {SYSTEM}
 => {applyTransitions(transferData(applyEnvTransitions(VAL, SYSTEM)))}
    in time period(SYSTEM)
 if containsEnvironment(SYSTEM) /\ VAL ;; VALS := allEnvAssignments(SYSTEM) .
```

where the function applyTransitions distributes to the thread objects in the state and is defined as follows for deterministic threads (whose transitions are defined using AADL's behavior annex):

```
ceq applyTransitions(
      < O : Thread | properties : Deterministic(true) ; PROPS,
                     features : PORTS,  currState : L1,  completeStates : LS,
                     variables : VAL,   behaviorRef : CR >)
  = if L2 in LS then < O : Thread | features : NEW-PORTS,   currState : L2,
                                    variables : NEW-VALUATION >
    else applyTransitions(< O : Thread | features: NEW-PORTS, currState : L2,
                                         variables : NEW-VALUATION >) fi
  if ((L1 -[GUARD]-> L2 {SL}) ; TRANSITIONS) := transitions(CR)
       /\ evalGuard(GUARD, PORTS, VAL)
       /\ transResult(NEW-PORTS, NEW-VALUATION) :=
            executeTransition(L1 -[GUARD]-> L2 {SL}, PORTS, VAL) .
```

Since the rewriting logic semantics of *Synchronous AADL* is *executable*, it defines an interpreter in Real-Time Maude for *Synchronous AADL* models, which has been integrated with the OSATE AADL tool as a plugin. We can use this plugin to perform both simulation and model checking verification of such models, as shown in Section 9.3 for the active standby system we have presented.

7. Hardware Description Language Semantics

The rewriting logic semantics project has been naturally extended from the level of programming languages to that of *hardware description languages* (HDLs). In this way, hardware designs written in an HDL can be both simulated and analyzed using the executable rewriting semantics of the HDL and tools like ELAN, CafeOBJ, or Maude. The first HDL to be given a rewriting logic semantics in Maude was ABEL [59]; this semantics was used not only for hardware designs, but also for hardware/software co-designs. An important new development has been the use of the rewriting logic semantics of an HDL for *generating sophisticated test inputs for hardware designs.* The point is that random testing can catch a good number of design errors, but uncovering deeper errors after random testing is hard and costly and requires a good understanding of the design to exercise complex computation sequences. The key insight, due to Michael Katelman, is that the rewriting semantics can be used *symbolically* to generate desired test inputs, not on a device's concrete states, but on states that are partly symbolic (contain logical variables) and partly concrete. This symbolic approach, first outlined in [146] and more fully developed in [147], has a number of unique features including: (i) the use of SAT solvers to symbolically solve Boolean constraints; (ii) support for user-guided random generation of partial instantiations; and (iii) a flexible *strategy language*, in which a hardware designer can specify in a declarative, high-level way the kind of test that needs to be generated. The effectiveness of this approach for generating sophisticated tests on real hardware designs, and for finding unknown bugs in such designs, has already been demonstrated for medium-sized Verilog designs, including the I^{2}C-Bus Master Controller, and a microprocessor design [147, 148].

But the value of the rewriting semantics of an HDL is not restricted to testing. For example, the recent Maude-based rewriting logic semantics of Verilog in [149] is arguably the most complete formal semantics to date, both in the sense of covering the largest subset of the language and in its faithful modeling of non-deteministic features. Besides being executable and supporting formal analysis, this semantics has uncovered several nontrivial bugs in various mature Verilog tools, and can serve as a practical and rigorous standard to ascertain what the correct behavior of such tools should be in complex cases.

A more exotic application of rewriting logic semantics, for which it is ideally suited due to its intrinsically concurrent nature, is that of *asynchronous hardware designs.* These are digital designs which do not have a global clock, so that different gates in a device can fire at different times. Such devices can behave correctly in much harsher environments (e.g., a satellite in outer space) and with much wider ranges of physical operating conditions than clocked devices. Asynchronous designs can be specified with the notation of *production rules*, which roughly speaking describe how each gate behaves when inputs to its wires are available. In [150] a rewriting logic semantics of asynchronous digital devices specified as sets of production rules is given and is realized in Maude. This is the first executable formal semantics of such devices we are aware of. It can be used both for simulation purposes and for model checking verification of small-sized devices (about 100 gates). An interesting challenge is how to scale up model checking for larger devices; this is nontrivial due to the large combinatorial explosion caused by their asynchronous behavior.

8. Abstract vs. Concrete Semantics and Static Analysis

In addition to helping with understanding and experimenting with language designs, a rewriting logic semantics can have several direct uses without having to change the semantics at all. Two such uses of unchanged semantics in the context of program verification are discussed in Sections 9 and 10. Nevertheless, there are program analysis needs where the desired information is not necessarily available in the code itself, or where the desired domain of analysis is not included in, and cannot be obtained from, the concrete domain in which the language semantics operates. In such cases, one can modify the concrete language semantics to operate within a target *abstract domain.* We next first show an overly simplified example, where the concrete semantics of IMP and IMP++ in Sections 3.1 and 3.2 are abstracted into type systems for the defined languages, which yield type checkers when executed. Then we discuss uses of similar but larger scale and more practical abstractions of rewrite logic semantics.

8.1. $\mathbb{K}$ Definition of a Type System for IMP++

The $\mathbb{K}$ semantics of IMP/IMP++ in Sections 3.1 and 3.2 can be used to execute even ill-typed IMP/IMP++ programs, which may be considered undesirable by some language designers. In this section we show how to define a type system for IMP/IMP++ using the very same $\mathbb{K}$ framework. The type system is defined like an (executable) semantics of the language, but one in the more abstract domain of types rather than in the concrete domain of integer and Boolean values.

The typing policy that we want to enforce on IMP/IMP++ programs is easy: all variables in a program have by default integer type and must be declared, arithmetic/Boolean operations are applied only on expressions of corresponding types, etc. Since programs and program fragments are now going to be rewritten into their types, we need to add to computations some basic types. Also, in addition to the computation to be typed, configurations must also hold the declared variables. Thus, we define the following (the "..." in the definition of K includes all the default syntax of computations, such as the original language syntax, $\curvearrowright$, freezers, etc.):

$$\begin{array}{rcl} K & ::= & \ldots \mid \mathit{int} \mid \mathit{bool} \mid \mathit{stmt} \mid \mathit{pgm} \\ \mathit{Configuration}^{\mathit{Type}}_{\mathrm{IMP++}} & \equiv & \langle \langle K \rangle_{\mathsf{k}} \; \langle \textbf{List}\{\mathit{Id}\} \rangle_{\mathsf{vars}} \rangle_{\top} \end{array}$$

Original language syntax	K Strict.	K Semantics
$AExp ::= Int$		$i \rightarrow int$
$\mid$ Id		$\langle \underset{int}{\underline{x}} \cdots\rangle_{\mathsf{k}} \langle\cdots x \cdots\rangle_{\mathsf{var}}$
$\mid$ $AExp$ `+` $AExp$	$[strict]$	int `+` $int \rightarrow int$
$\mid$ $AExp$ `/` $AExp$	$[strict]$	int `/` $int \rightarrow int$
$\mid$ `++` Id		$\langle \underset{int}{\underline{\texttt{++}\,x}} \cdots\rangle_{\mathsf{k}} \langle\cdots x \cdots\rangle_{\mathsf{var}}$
$BExp ::= AExp$ `<=` $AExp$	$[strict]$	int `<=` $int \rightarrow bool$
$\mid$ `not` $BExp$	$[strict]$	`not` $bool \rightarrow bool$
$\mid$ $BExp$ `and` $BExp$	$[strict]$	$bool$ `and` $bool \rightarrow bool$
$Stmt ::=$ `skip`		`skip` $\rightarrow stmt$
$\mid$ Id `:=` $AExp$	$[strict(2)]$	$\langle \underset{stmt}{\underline{x \,\texttt{:=}\, int}} \cdots\rangle_{\mathsf{k}} \langle\cdots x \cdots\rangle_{\mathsf{var}}$
$\mid$ $Stmt$ `;` $Stmt$	$[strict]$	$stmt$ `;` $stmt \rightarrow stmt$
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$	$[strict]$	`if` $bool$ `then` $stmt$ `else` $stmt$ $\rightarrow stmt$
$\mid$ `while` $BExp$ `do` $Stmt$	$[strict]$	`while` $bool$ `do` $stmt \rightarrow stmt$
$\mid$ `print` $AExp$	$[strict]$	`print` $int \rightarrow stmt$
$\mid$ `halt`		`halt` $\rightarrow stmt$
$\mid$ `spawn` $Stmt$	$[strict]$	`spawn` $stmt \rightarrow stmt$
$Pgm ::=$ `var` $\mathbf{List}\{Id\}$ `;` $Stmt$		$\langle \underset{s \curvearrowright pgm}{\underline{\texttt{var}\ xl\ ;\ s}}\rangle_{\mathsf{k}} \langle \underset{xl}{\underline{\cdot}} \rangle_{\mathsf{vars}}$ $stmt \curvearrowright pgm \rightarrow pgm$

Figure 6: K type system for IMP++ (and IMP)

Figure 6 shows the IMP/IMP++ type system as a $\mathbb{K}$ system over such configurations. Constants reduce to their types, and types are straightforwardly propagated through each language construct. Note that almost each language construct is strict now, because we want to type all its arguments in almost all cases in order to apply the typing policy of the construct. Two constructs are exceptional, namely, increment and assignment. The typing policy of these constructs is that they take precisely a variable and not something that types to an integer. If we defined, e.g., the assignment strict and with rule $int \,\texttt{:=}\, int \rightarrow stmt$, then our type system would allow ill-formed programs like `x+y := 0`. Note how we defined the typing policy of programs `var` xl ; s: the declared variables xl are stored into the $\langle\ldots\rangle_{\mathsf{vars}}$ cell (which is expected to initially be empty) and the statement is scheduled for typing (using a structural rule), placing a "reminder" in the computation that the *pgm* type is expected; once/if the statement is correctly typed, the type *pgm* is generated.

8.2. Examples of Abstract Rewriting Logic Semantics

We briefly discuss three practical uses of abstract rewriting logic semantics.

8.2.1. C Pluggable Policies.

Many programs make implicit assumptions about data. Common examples include assumptions about whether variables have been initialized or can only contain non-null references. Domain-specific examples are also common; a compelling example is units of measurement, used in many scientific computing applications, where different variables and values are assumed to have specific units at specific times/along specific execution paths. These implicit assumptions give rise to implicit domain *policies*, such as requiring assignments to non-null pointers to also be non-null, or requiring two operands in an addition operation to have compatible units of measurement.

Mark Hills et al. [151] propose a framework for *pluggable policies* for C which allows these implicit policies to be made explicit and checked. The core of the framework is a shared annotation engine and parser, allowing annotations in multiple policies to be inserted by developers as comments in C programs, and a shared abstract rewriting logic semantics of C designed as a number of reusable modules that allow for new policies to be quickly developed and plugged in. For instance, a case study for checking non-null references was developed in under two days; another case study for checking units of measurement reuses the shared abstract semantics and only adds domain knowledge [151].

8.2.2. Polymorphic Type Inference.

The technique in Section 8.1 for defining type systems using $\mathbb{K}$ is very general and has been used to define more complex type systems, such as higher-order polymorphic ones by Ellison et al. [152]. The $\mathbb{K}$ definition of the type system in [152] is more declarative and thus cleaner and easier to understand than alternative algorithmic definitions. Moreover, the $\mathbb{K}$ definition is formal, so it is amenable for formal reasoning. Interestingly, as shown in [152], the resulting $\mathbb{K}$ definition, when compiled to and executed using Maude, was faster than algorithmic implementations of the same type system found on the Internet as teaching material. In fact, experiments in [152] show that it was comparable to state of the art implementations of type inferencers in conventional functional languages! For example, it was only about twice as slow on average than that of OCaml, and had average times comparable, or even better than those of Haskell ghci and SML/NJ.

8.2.3. Security Policy Checking.

An elegant application of a programming language's *abstract* rewriting logic semantics to Java code security is presented by Alba-Castro et al. in [153, 154] as part of their rewriting-logic-semantics-based approach to proof carrying code. The key idea is to use an abstract rewriting logic semantics of Java that correctly approximates security properties such as *noninterference* (that is, the specification of what objects should not have any effects on other objects according to a stated security policy [155]), and *erasure* (a security policy that mandates that secret data should be removed after its intended use). Since the abstract rewriting semantics is finite-state, it supports the automatic creation of certificates for noninterference and erasure properties of Java programs that are independently checkable and small enough to be practical.

9. Model Checking Verification

Once a programming language or system is defined as a rewrite theory, one can use any general-purpose tools and techniques for rewriting logic to obtain tools and techniques specialized for the defined programming language or system. We have reported in the past on the use of Maude's general purpose LTL model checking capabilities to obtain model checkers specialized for various concurrent programming languages, including Java and the JVM (see, e.g., [2, 46, 45]). In this paper we report on recent results on using a rewriting logic semantics of a language in Maude, or of a real-time language in Real-Time Maude, *directly* to model check programs in the given language. Specifically, Section 9.1 discusses such model checking for C programs, Section 9.2 does so for Ptolemy II models, and Section 9.3 covers the model checking of *Synchronous AADL* models.

9.1. Model Checking Verification of C Programs

We present some new model checking experiments performed in the context of the C definition discussed in Section 4. We thank Chucky Ellison for extending his C semantics with concurrency primitives and for conducting these experiments. A more detailed presentation of these can be found in [97].

The C semantics in Section 4 can be extended to include semantics for concurrency primitives like "spawn", "sync", "lock", and "unlock". The former is used to dynamically spawn a new execution thread, "sync" waits for all of the other threads to die before continuing, and "lock" and "unlock" synchronize threads on memory locations (similar to Java locking on references). When formalizing the semantics of C, we did not plan to introduce concurrency. Despite that, as hoped for, the existing rules were left unchanged upon adding configuration support and the semantics of threads.

Dekker's Algorithm. We now take a look at the classical Dekker's algorithm, in order to explore thread interleavings.

```
void dekker1(void) {                            void dekker2(void) {
  flag1 = 1;  turn = 2;                           flag2 = 1;  turn = 1;
  while((flag2 == 1) && (turn == 2)) ;            while((flag1 == 1) && (turn == 1)) ;
  critical1();                                    critical2();
  flag1 = 0;                                      flag2 = 0;
}                                               }
```

These two functions get called by the two threads respectively to ensure mutual exclusion of the calls to `criticaln()`. In the program we used for testing, these threads each contain infinite loops while the function `main()` waits on a `sync()`. Thus, the program never terminates.

To test the mutual exclusion property, we model check the following LTL formula: $\Box\neg(\mathrm{enabled}(\mathit{critical1}) \wedge \mathrm{enabled}(\mathit{critical2}))$, stating that the two critical sections can never be called at the same time. Applying this formula to our program yields “`result Bool: true`”, in 400ms. If we break the algorithm by changing a `while` to an `if`, the tool instead returns a list of rules, together with the resulting states, that represent a counterexample.

Dining Philosophers. Another classic example is the dining philosophers problem.

```
void philosopher( int n ) {
  while(1) {
    // Hungry: obtain chopsticks
    if ( n % 2 == 0 ) {  // Even number: Left, then right
      lock(&chopstick[(n+1) % NUM_PHILOSOPHERS]);
      lock(&chopstick[n]);
    } else {  // Odd number: Right, then left
      lock(&chopstick[n]);
      lock(&chopstick[(n+1) % NUM_PHILOSOPHERS]);
    }
    // Eating
    // Finished Eating: release chopsticks
    unlock(&chopstick[n]);
    unlock(&chopstick[(n+1) % NUM_PHILOSOPHERS]);
    // Thinking
  }
}
```

The above code shows a solution to the dining philosophers that has even-numbered philosophers picking up their left chopstick first, while odd-numbered philosophers pick up their right chopstick first. This strategy ensures that there is no deadlock. We can use Maude’s search command to verify that there is no deadlock simply by searching for final states. Here are the results:

	No Deadlock		With Deadlock	
n	number of states	time (s)	number of states	time (s)
1	19	0.1	–	–
2	92	0.8	63	0.6
3	987	14.0	490	7.2
4	14610	293.5	5690	119.8
5	288511	8360.3	84369	2376.5

In the “No Deadlock” column we see the results for the code above. We were able to verify that with this algorithm, there were no deadlocks for up to five philosophers. In the “With Deadlock” column, we altered the code so that all philosophers would try to pick up their left chopstick first. For this algorithm, we were able to find counterexamples showing that the program has deadlocks.

While the classic programs above are toy examples, which are far from the complexity of real-life software, we believe that they are sufficient to show that a programming language semantics can be more than a “useless academic intellectual exercise”. The well-known state-space explosion of model checking cannot be avoided, no matter whether one uses a formal semantics of the language or not, but one should note that this is a problem of model checking and not of using a formal semantics for model checking. Also, there are well-known techniques to address the state explosion problem, like partial-order reduction, which can and have also been applied in the context of rewriting logic semantics

[52]. And one can use an *abstract semantics* (Section 8) as the basis of the model checker to make it more scalable. The next section shows another use of rewriting logic semantics of programming languages, for deductive program verification.

9.2. Model Checking Verification of Ptolemy II Models

We show here how the rewriting logic semantics of Ptolemy II DE models presented in Section 6.2 can, thanks to the integration of that semantics in Real-Time Maude within Ptolemy II, be used *from within Ptolemy II* to model check temporal logic properties of DE models. In particular, using the Real-Time Maude plugin, a Ptolemy II user only needs knowledge of temporal logic to specify such properties and does not need any knowledge of Real-Time Maude. This is because the plugin provides a simple property specification language in which state predicates can be defined in terms of the values of state variables of the different actors in the given Ptolemy II model. Let us illustrate all this on the traffic system example presented in Section 6.2.

The following *timed* CTL property states that the car light will turn yellow, *and only yellow*, within 1 time unit of a failure:

```
AG (('HierarchicalTrafficLight . 'Decision | port 'Error is present)
   => AF[<= 1] ('HierarchicalTrafficLight | 'Cyel = 1, 'Cgrn = 0, 'Cred = 0))
```

Note how the property language allows using state predicates that refer to state variables in various actors of the Ptolemy II model, so that no knowledge of the underlying Real-Time Maude is required to specify this property.

As shown in Figure 7, model checking this property from within Ptolemy II returns a previously unknown counter-example which shows that, after a failure, the car light may show red or green in addition to blinking yellow.

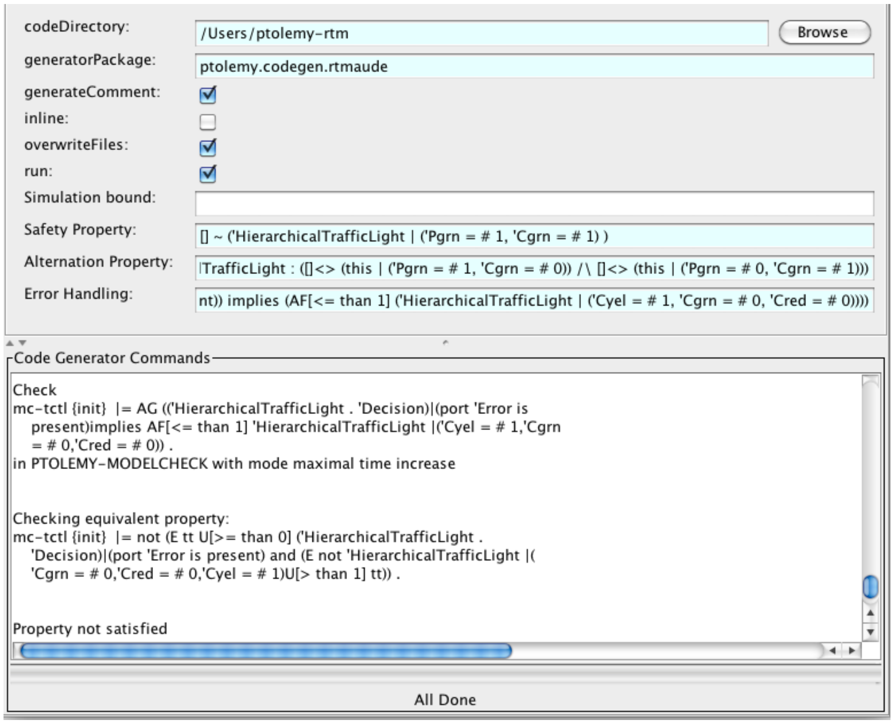
9.3. Model Checking Verification of Synchronous AADL Models

We illustrate here how the formal executable semantics of *Synchronous AADL* summarized in Section 6.2 can be used, taking advantage of Real-Time Maude model checking capabilities, to verify formal properties of a model. For the Active Standby System described in Section 6.2, the key properties that should be verified according to [144] are:

R_1: Both sides should agree on which side is active (provided neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).

R_2: A side that is not fully available should not be the active side if the other side is fully available (again, provided neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).

R_3: The pilot can always change the active side (except if a side is failed or the availability of a side has changed).


codeDirectory: /Users/ptolemy-rtm

generatorPackage: ptolemy.codegen.rtmaude
generateComment: ☑
inline: □
overwriteFiles: ☑
run: ☑
Simulation bound:
Safety Property: [] ~ ('HierarchicalTrafficLight | ('Pgrn = # 1, 'Cgrn = # 1))
Alternation Property: |TrafficLight : ([]<> (this | ('Pgrn = # 1, 'Cgrn = # 0)) /\ []<> (this | ('Pgrn = # 0, 'Cgrn = # 1)))
Error Handling: nt)) implies (AF[<= than 1] ('HierarchicalTrafficLight | ('Cyel = # 1, 'Cgrn = # 0, 'Cred = # 0))))

Code Generator Commands

```
Check
mc-tctl {init}  |= AG (('HierarchicalTrafficLight . 'Decision)|(port 'Error is
   present)implies AF[<= than 1] 'HierarchicalTrafficLight |('Cyel = # 1,'Cgrn
   = # 0,'Cred = # 0)) .
in PTOLEMY-MODELCHECK with mode maximal time increase


Checking equivalent property:
mc-tctl {init}  |= not (E tt U[>= than 0] ('HierarchicalTrafficLight .
   'Decision)|(port 'Error is present) and (E not 'HierarchicalTrafficLight |(
   'Cgrn = # 0,'Cred = # 0,'Cyel = # 1)U[> than 1] tt)) .


Property not satisfied
```

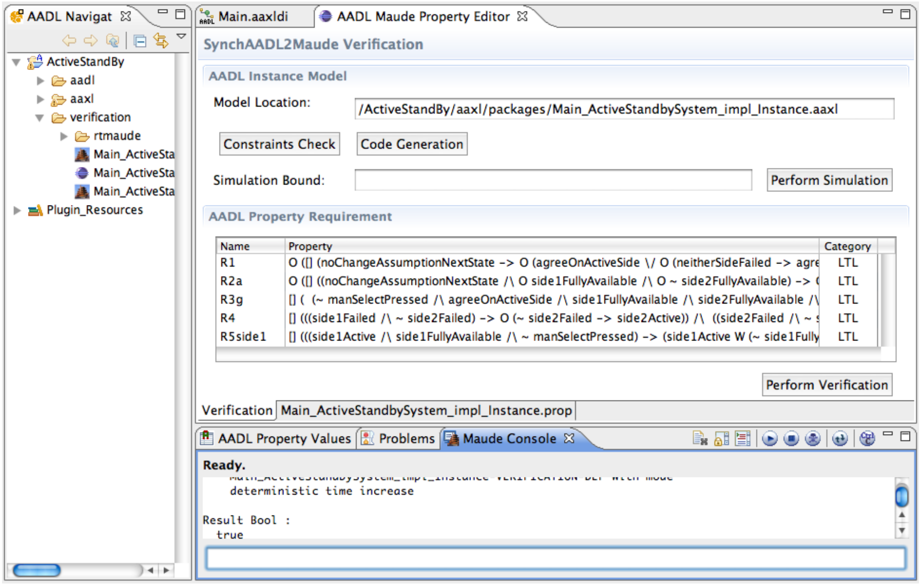
All Done

Figure 7: Dialog window for the Real Time Maude code generation and analysis.

R_4: If a side is failed the other side should become active.

R_5: The active side should not change unless the availability of a side changes, the failed status of a side changes, or manual selection is selected by the pilot.

To verify such properties, a *Synchronous AADL* user should be reasonably familiar with temporal logic. However, since in a way similar to the Ptolemy II plugin discussed in Section 9.2: (i) the formal executable semantics of *Synchronous AADL* in Real-Time Maude has been integrated as the *SynchAADL2Maude* plugin within the OSATE AADL tool, and (ii) a simple language to define *state predicates* of a *Synchronous AADL* system *in terms of variables in the model itself* is also offered to the user for defining properties, such a user *does not need to be familiar with the underlying Real-Time Maude* and can specify and verify temporal logic properties *in terms of the Synchronous AADL model alone* (see [142] for details). For example, for the Active Standby model we can specify the relevant state predicates this way and then verify (a refined version of) properties R_1–R_5 from within the OSATE plugin, as shown in Figure 8.



SynchAADL2Maude Verification

AADL Instance Model

Model Location: /ActiveStandBy/aaxl/packages/Main_ActiveStandbySystem_impl_Instance.aaxl

Simulation Bound:

AADL Property Requirement

Name	Property	Category
R1	O ([] (noChangeAssumptionNextState -> O (agreeOnActiveSide \/ O (neitherSideFailed -> agre	LTL
R2a	O ([] ((noChangeAssumptionNextState /\ O side1FullyAvailable /\ O ~ side2FullyAvailable) -> (	LTL
R3g	[] ((~ manSelectPressed /\ agreeOnActiveSide /\ side1FullyAvailable /\ side2FullyAvailable /\	LTL
R4	[] (((side1Failed /\ ~ side2Failed) -> O (~ side2Failed -> side2Active)) /\ ((side2Failed /\ ~ s	LTL
R5side1	[] (((side1Active /\ side1FullyAvailable /\ ~ manSelectPressed) -> (side1Active W (~ side1Fully	LTL

Ready.

```
    deterministic time increase

Result Bool :
  true
```

Figure 8: SynchAADL2Maude verification window in OSATE.

10. Deductive Verification and Matching Logic

As discussed above, one of the major advantages of giving a rewriting logic semantics to a language is that one can use it not only to obtain a reference implementation of the language, but also to formally analyze programs in the defined language using general-purpose tools developed for rewriting logic, such as Maude's model checker. Moreover, the original rewriting logic semantics of the language is used unchanged for model checking or other similar analyses, which is not only immensely convenient but also offers a high confidence in the results of the analysis (because it excludes the problem of implementing a wrong language semantics in the analyzer). One question, however, still remains unanswered: can we use the language semantics, also unchanged, in a program logic fashion, that is, for deductive verification of programs?

Early work in this direction includes two Hoare logic provers that use directly the rewriting logic semantics of a Pascal like-language and of a fragment of Java and the Maude ITP [53, 58]. Furthermore, the rewriting logic semantics of Java was used in [55] to automatically validate the inference rules of a Java verification tool. All these early efforts were still based on an additional Hoare logic semantics of the target languages, but used the rewriting logic semantics of the language to validate or automate the application of the Hoare logic proof rules. In the remainder of this section we report on an alternative approach, which needs no other semantics of the language for verification purposes. It uses precisely the rewriting semantics of the language and nothing else for deductive verification.

Matching logic [86, 85, 84, 83, 82] is a new program verification logic, which builds upon rewriting logic semantics. Matching logic specifications are constrained symbolic program configurations, called *patterns*, which can be *matched* by concrete configurations. By building upon an executable semantics of the language and allowing specifications to directly refer to the structure of the configuration, matching logic has at least three benefits: (1) one's familiarity with the formalism reduces to one's familiarity with the formal semantics of the language, that is, with the language itself; (2) the verification process proceeds the same way as the program execution, making debugging failed proof attempts manageable because one can always see the "current configuration" and "what went wrong", almost like in a debugger; and (3) nothing is lost or distorted in translation, that is, there is no gap between the language definition and its verifier. Moreover, direct access to the structure of the configuration facilitates defining sub-patterns that one may reason about, such as disjoint lists or trees in the heap, as well as supporting framing in various components of the configuration at no additional cost.

To use matching logic for program verification, one must know the structure of the configurations that are used in the executable language semantics. For example, the configuration of some language may contain, besides the code itself, an environment, a heap, stacks, synchronization resources, etc. The configuration of C (Section 4 and [97]), e.g., consists of more than 70 cells, each containing either other cells or some piece of semantic information. Of course, thanks to the modular structure of configurations, only the relevant cells (typically just a few) and only the relevant cell contents (typically very small) need to be mentioned in a given matching logic specification. Matching logic specifications, or patterns, allow one to refer directly to the configuration of the program. Moreover, we can use logical variables and thus combine the desired configuration structure with first-order constraints. For example, the pattern

$$\begin{array}{l}\langle \quad \langle \beta,\ I\rangle_{\text{in}}\ \ \langle \mathsf{x} \mapsto x,\ \mathsf{i} \mapsto i,\ \mathsf{n} \mapsto n,\ E\rangle_{\text{env}}\ \ \langle \mathsf{list}(x,\alpha),\ H\rangle_{\text{heap}}\ \ C \quad \rangle_{\text{config}} \\ \quad \wedge\ \ i \le n \quad \wedge \quad |\beta| = n - i \quad \wedge \quad A = \mathit{rev}(\alpha)@\beta\end{array}$$

specifies the set of configurations where program variables x, i and n are bound in the environment to some respective values x, i, and n, such that $i \le n$, the input buffer contains a sequence β of size $n - i$, and the heap contains a linked list starting with pointer x comprising the sequence of elements α such that the sequence A is the reverse of the sequence α concatenated with β. Here A is a free variable of type sequence of elements. The other variables play the role of cell frames: I is a variable matching the rest of the input cell, E matches the rest of the environment, H the rest of the heap, and C the rest of the configuration. Note that nothing special needs to be done for framing in matching logic (that is, framing is a special case of the more general principle of matching).

A major benefit of matching logic is that it can be used to turn an executable semantics into a program logic without any change to the original semantics. The idea is that the executable semantics can be regarded as a set of rewrite rules between matching logic patterns, and one can use first-order reasoning over patterns to turn the pattern resulting from the application of some rule into a pattern that the next rule expects to match. This way, one can derive rewrite rules from other rewrite rules, using matching logic reasoning as a mechanism to rearrange configurations so that rewrite rules can match and apply.

All this can be formalized as a *language-independent* and *sound* proof system for deriving reachability rules between matching logic patterns [86, 84]. We stress the language independence of the matching logic proof system, because this clearly distinguishes it from Hoare logic. As we know, one has to define a specific Hoare logic for each given language in order to do formal deductive program verification. This is a very tedious and error prone process, because defining a Hoare logic is not as intuitive and straightforward as defining an operational semantics. Consequently, the current state of the art in mechanical deductive verification is to have both an operational (trusted) semantics and a Hoare logic for a language, and then prove the soundness of the latter based on the former. In matching logic one needs no additional semantics, axiomatic or of any other nature, and no tedious soundness proofs need to be done for each language separately. This is because matching logic deduction uses the same rewriting semantic rules of the language as axioms, and the proof system allowing to derive rewrites between patterns is completely agnostic about the rewrite rules giving the language semantics. The matching logic proof system includes both rules borrowed from rewriting logic and rules inspired from the language-independent rules of Hoare logic, plus one very specific rule, called *Circularity*. The Circularity rule is coinductive in nature and captures in a language-independent way the invariant-like nature of language constructs that have circular behaviors (loops, recursion, jumps, etc.). As shown in [85], the Circularity rule is powerful enough to allow any Hoare logic proof derivation to be mechanically translated into a proof derivation based on the matching logic proof system.

With the help of Andrei Ştefănescu, we implemented a proof-of-concept matching logic verifier for a fragment of C, called MATCHC, which can be downloaded and executed online at `http://fsl.cs.uiuc.edu/ml`. MATCHC builds upon an executable rewrite-based semantics of this fragment of C, extending it (unchanged) with semantics for pattern specifications. Both the executable semantics and the verifier are implemented using the $\mathbb{K}$ framework (see Section 3).

Figure 9 shows a C program verified using MathC. The `main()` function reads `n` from the standard input and then calls `readWriteBuffer(n)`. Then `readWriteBuffer(n)` reads from the standard input `n` elements and allocates a linked list putting each element at the top of the list, followed by traversing the linked list and printing each element while deallocating the list nodes. This way, we end up with the reversed sequence of elements printed to the the standard output and with the heap unchanged. There are four types of annotations in this program: (1) *assumptions*, which allow one to assume a certain pattern for the remaining program; (2) *assertions*, which generate matching logic proof obligations, namely, that the current pattern implies the asserted pattern; (3) *rules*, which give the claimed $\mathbb{K}$ semantics of the subsequent piece of code; and (4) *invariants*, which are patterns that should hold at each loop iteration.

Some explanations regarding MatchC's notation are necessary. MatchC an-

```
#include <stdlib.h>
#include <stdio.h>

struct listNode { int val; struct listNode *next; };

void readWriteBuffer(int n)
/*@ rule <k> $ => return;...</k>  <in> A => epsilon...</in>  <out>...epsilon => rev(A) </out>
    if n = len(A) */
{
  int i;  struct listNode *x;
  i = 0;  x = 0;
  /*@ inv <in> ?B...</in> <heap>...list(x)(?A)...</heap>
          /\ i <= n /\ len(?B) = n - i /\ A = rev(?A) @ ?B */
  while (i < n) {
    struct listNode *y;
    y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1;
  }

  //@ inv <out>...?A </out> <heap>...list(x)(?B)...</heap> /\ A = rev(?A @ ?B)
  while (x) {
    struct listNode *y;
    y = x->next;
    printf("%d ",x->val);
    free(x);
    x = y;
  }
}

void main() {
  int n;
  //@ assume <in> [5, 1, 2, 3, 4, 5] </in>  <out> epsilon </out>
  scanf("%d", &n);
  readWriteBuffer(n);
  //@ assert <in> epsilon </in> <out> [5, 4, 3, 2, 1] </out>
}
```

Figure 9: C program making use of the I/O and the heap, verified using MatchC.

notations are introduced like C comments starting with @, so they are ignored by C compilers. We use an XML-like notation to specify when cells start and when they end. We use the usual rewriting relation "`=>`" for the in-place rewriting within 𝕂 rules. The "`$`" symbol that appears in the computation cell of a rule stands for the subsequent statement (the function body, in our case here). Fourth, to avoid writing quantifiers, variables starting with a question mark are existentially quantified over the pattern. Fifth, we use ellipses to state that the corresponding cell is open in that direction, which can be regarded as an abbreviation for using a fresh variable; for example, "`<in> ?B ...</in>`" in the invariant of the first loop abbreviates "`<in> ?B, ?E </in>`". Finally, to avoid writing the environment cell all the time, MatchC allows users to refer directly to program variables in patterns; this avoids having to add a binding of the program variable to a logical variable in the environment cell and then using the logical variable throughout the pattern.

The rule giving the semantics of `readWriteBuffer(n)` states that this function returns nothing ("`$ => return;`", that is, its body behaves as if it returns) and takes a sequence A of length `n` (see the condition "`n = len(A)`") from the beginning of the input cell (“`<in> A => epsilon...<in>`”) and places it reversed at the end of the output cell (“`<out>...epsilon => A </out>`”). Since we have a rewrite-based semantics, the fact that no other cells are mentioned implicitly means that *nothing else is modified by this function*, including the heap. The invariant of the first loop is exactly the pattern that we discussed at the beginning of this section. The invariant of the second loop is similar, but dual. We do not show the axiom (matching logic formula) governing the list pattern in the heap cell; the interested reader can check [86, 83, 82]. Nevertheless, since `x` is null at the end of the second loop, it follows that the list it points to is empty, so the heap changes by the first loop will be cleaned by the end of the second.

MatchC verifies the program in Figure 9 in about 100 milliseconds:

```
Compiling program ... DONE! [0.311s]
Loading Maude ....... DONE! [0.209s]
Verifying program ... DONE! [0.099s]
Verification succeeded! [82348 rewrites, 4 feasible and 2 infeasible paths]
Output: 5 4 3 2 1
```

Dozens of C programs have been verified using MatchC, most known to be problematic to verify using existing approaches based on Hoare logic or extensions of it. The list of verified programs includes:

- undefined programs according to the semantics of C, which should not be provably correct (unfortunately, such programs are “proved” correct by some existing program verifiers which, obviously, are not based on an executable and thus testable semantics of the programming language);

- conventional Hoare logic programs which make no use of the heap or other cells in the configuration except for the environment; programs using the input/output;

- list-manipulating programs taken over from separation logic tools, making intensive use of the heap;

- tree-manipulating programs, including search trees such as binary-search trees and AVL trees;

- graph-manipulating programs, including the famous Schorr-Waite graph marking algorithm.

Two factors guided us in choosing these programs and in our MatchC verification effort: (1) proving functional correctness (as opposed to just memory safety), and (2) doing so automatically (the user only provides the specifications). The Schorr-Waite graph marking algorithm [156] computes all the nodes in a graph that are reachable from a set of starting nodes. To achieve that, it visits the graph nodes in depth-first search order, by reversing pointers on the way down, and then restoring them on the way up. Its main application is in garbage collection. The Schorr-Waite algorithm presents considerable verification challenges [157, 158]. We analyzed the algorithm itself as originally given for graphs, and a simplified version in which the graph is a tree. For both cases we proved that a node is marked if and only if it is reachable from the set of initial nodes, and that the graph does not change. Most of these examples are proved in milliseconds and do not require SMT support. The source code of MatchC, as well as an online interface allowing one to verify and experiment with all C programs discussed here, or to introduce new ones, is publicly available from the matching logic web page at `http://fsl.cs.uiuc.edu/ml`.

11. Conclusions and Future Work

We have given a progress report on the rewriting logic semantics project. Our main goal has been to show how research in this area is closing the gap between theory and practice by supporting executable semantic definitions that scale up to real languages at the three levels of software modeling languages, programming languages, and HDLs, and with features such as concurrency and real-time semantics. We have also shown how such semantic definitions can be *directly* used as a basis for interpreters and for sophisticated program analysis tools, including static analyzers, model checkers, and program proving tools.

Although reasonably efficient *interpreters* can be currently generated from rewriting logic specifications, one important future challenge is the automatic generation from language definitions of high-performance language implementations that are correct by construction. Another area that should be further developed is that of *meta-reasoning* methods, to prove formal properties not about programs, but about entire language definitions. A third promising future research direction is exploring the systematic interplay between abstract semantics and model checking, as well as the systematic application of state space reduction techniques in the model checking of programs from their rewriting logic language definitions; the overall goal is achieving a high degree of *scalability* in model checking analyses, with a wide spectrum of analysis choices ranging from model checking of programs according to their concrete semantics to various forms of static analysis based on different kinds of abstract semantics.

Yet another exciting research direction is the development of generic *theorem prover generators*, which, like in the case of the generic model checker facility provided by Maude, take an entire language definition as input, and produce an efficient, full-fledged theorem prover for the given language. For example, a prover like the MatchC tool could in this way be derived from the C semantic definition, and so could also provers for many other languages.

Acknowledgments.. We thank the organizers of FCT 2011 for giving us the opportunity of presenting these ideas in preliminary form at the FCT 2011 conference, and both them and the FCT 2011 participants for their comments, which have helped us develop the expanded and improved version presented here. We also thank all the researchers involved in the rewriting logic semantics project for their many contributions, which we have tried to summarize in this paper without any claims of completeness. This research has been partially supported by NSF Grants CNS 0834709, CCF 0905584, CCF 0916893, and

CCF 1218605, by AFOSR Grant FA8750-11-2-0084, by NSA contract H98230-10-C-0294, by (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010, and by the Boeing Grant C8088.

References

[1] J. Meseguer, G. Roşu, Rewriting logic semantics: From language specifications to formal analysis tools, in: Proc. IJCAR'04, Vol. 3097 of LNAI, Springer, 2004, pp. 1–44.
URL `http://dx.doi.org/10.1007/978-3-540-25984-8_1`

[2] J. Meseguer, G. Roşu, The rewriting logic semantics project, Theoretical Computer Science 373 (2007) 213–237.

[3] T. F. Şerbănuţă, G. Roşu, J. Meseguer, A rewriting logic approach to operational semantics, Information and Computation 207 (2) (2009) 305–340.
URL `http://dx.doi.org/10.1016/j.ic.2008.03.026`

[4] J. Meseguer, G. Rosu, The rewriting logic semantics project: A progress report, in: O. Owe, M. Steffen, J. A. Telle (Eds.), FCT, Vol. 6914 of Lecture Notes in Computer Science, Springer, 2011, pp. 1–37.

[5] G. D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60-61 (2004) 17–139, Previously published as technical report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[6] G. Kahn, Natural semantics, in: Proc. STACS'87, Vol. 247 of LNCS, Springer, 1987, pp. 22–39.

[7] P. D. Mosses, Modular structural operational semantics, J. Log. Algebr. Program. 60–61 (2004) 195–228.

[8] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.

[9] M. Felleisen, D. P. Friedman, Control operators, the SECD-machine, and the λ-calculus, in: 3rd Working Conference on the Formal Description of Programming Concepts, Denmark, 1986, pp. 193–219.

[10] G. Berry, G. Boudol, The chemical abstract machine, Theoretical Computer Science 96 (1) (1992) 217–248.

[11] M. Wand, First-order identities as a defining language, Acta Informatica 14 (1980) 337–357.

[12] J. A. Goguen, K. Parsaye-Ghomi, Algebraic denotational semantics using parameterized abstract modules, in: J. Diaz, I. Ramos (Eds.), Formalizing Programming Concepts, Springer-Verlag, 1981, pp. 292–309, lNCS, Volume 107.

[13] M. Broy, M. Wirsing, P. Pepper, On the algebraic definition of programming languages, ACM TOPLAS 9 (1) (1987) 54–99.

[14] P. D. Mosses, Unified algebras and action semantics, in: Proc. Symp. on Theoretical Aspects of Computer Science, STACS'89, Springer LNCS 349, 1989, pp. 17–35.

[15] J. Goguen, G. Malcolm, Algebraic Semantics of Imperative Programs, MIT Press, 1996.

[16] A. van Deursen, J. Heering, P. Klint, Language Prototyping: An Algebraic Specification Approach, World Scientific, 1996.

[17] D. Scott, Outline of a mathematical theory of computation, in: Proceedings, Fourth Annual Princeton Conference on Information Sciences and Systems, Princeton University, 1970, pp. 169–176, also appeared as Technical Monograph PRG 2, Oxford University, Programming Research Group.

[18] D. Scott, C. Strachey, Toward a mathematical semantics for computer languages, in: Microwave Research Institute Symposia Series, Vol. 21: Proc. Symp. on Computers and Automata, Polytechnical Institute of Brooklyn, 1971.

[19] D. A. Schmidt, Denotational Semantics – A Methodology for Language Development, Allyn and Bacon, Boston, MA, 1986.

[20] P. D. Mosses, Denotational semantics, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B, Chapter 11, North-Holland, 1990.

[21] D. P. Friedman, M. Wand, C. T. Haynes, Essentials of Programming Languages, 2nd Edition, MIT Press, Cambridge, MA, 2001.
URL `http://www.cs.indiana.edu/eopl/`

[22] B. Pierce, Types and Programming Languages, MIT Press, 2002.

[23] M. Kaufmann, P. Manolios, J. S. Moore, Computer-Aided Reasoning: ACL2 Case Studies, Kluwer Academic Press, 2000.

[24] E. Moggi, An abstract view of programming languages, Tech. Rep. ECS-LFCS-90-113, Edinburgh University, Dept. of Computer Science (June 1989).

[25] P. Wadler, The essence of functional programming, in: Proc. POPL '92, ACM Press, New York, NY, USA, 1992, pp. 1–14. `doi:http://doi.acm.org/10.1145/143165.143169`.

[26] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: Proc. POPL'95, ACM Press, 1995, pp. 333–343. `doi:http://doi.acm.org/10.1145/199448.199528`.

[27] F. Pfenning, C. Elliott, Higher-order abstract syntax, in: Proc. PLDI'88, ACM Press, 1988, pp. 199–208.

[28] R. Harper, F. Honsell, G. D. Plotkin, A framework for defining logics, Journal of the ACM 40 (1) (1993) 143–184.

[29] G. Nadathur, D. Miller, An overview of λProlog, in: K. Bowen, R. Kowalski (Eds.), Fifth Int. Joint Conf. and Symp. on Logic Programming, The MIT Press, 1988, pp. 810–827.

[30] D. Miller, Representing and reasoning with operational semantics, in: Proc. IJCAR'06, Vol. 4130 of LNCS, 2006, pp. 4–20.

[31] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, CENTAUR: The system, in: Software Development Environments (SDE), 1988, pp. 14–24.

[32] D. Clément, J. Despeyroux, L. Hascoet, G. Kahn, Natural semantics on the computer, in: K. Fuchi, M. Nivat (Eds.), Proceedings, France-Japan AI and CS Symposium, ICOT, 1986, pp. 49–89, also, Information Processing Society of Japan, Technical Memorandum PL-86-6.

[33] K. Slonneger, B. L. Kurtz, Formal Syntax and Semantics of Programming Languages, Addison-Wesley, 1995.

[34] Y. Gurevich, Evolving algebras 1993: Lipari Guide, in: E. Börger (Ed.), Specification and Validation Methods, Oxford University Press, 1994, pp. 9–37.

[35] R. F. Stärk, J. Schmid, E. Börger, Java and the Java Virtual Machine: Definition, Verification, Validation, Springer, 2001.

[36] C. Braga, Rewriting logic as a semantic framework for modular structural operational semantics, Ph.D. thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brazil (2001).

[37] A. Verdejo, N. Martí-Oliet, Implementing CCS in Maude 2, in: F. Gadducci, U. Montanari (Eds.), Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS, Elsevier, 2002.

[38] P. Thati, K. Sen, N. Martí-Oliet, An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0, in: F. Gadducci, U. Montanari (Eds.), Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS, Elsevier, 2002.

[39] M.-O. Stehr, C. Talcott, PLAN in Maude: Specifying an active network programming language, in: F. Gadducci, U. Montanari (Eds.), Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications, Vol. 117, ENTCS, Elsevier, 2002.

[40] J. Meseguer, Software specification and verification in rewriting logic, in: M. Broy, M. Pizka (Eds.), Models, Algebras, and Logic of Engineering Software, NATO Advanced Study Institute, Marktoberdorf, Germany, July 30 – August 11, 2002, IOS Press, 2003, pp. 133–193.

[41] A. Verdejo, Maude como marco semántico ejecutable, Ph.D. thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain (2003).

[42] F. Chen, G. Roşu, R. P. Venkatesan, Rule-based analysis of dimensional safety, in: Proc. RTA'03, Vol. 2706 of LNCS, 2003, pp. 197–207.

[43] G. Roşu, R. P. Venkatesan, J. Whittle, L. Leustean, Certifying optimality of state estimation programs, in: Computer Aided Verification (CAV'03), Springer, 2003, pp. 301–314, lNCS 2725.

[44] A. Verdejo, N. Martí-Oliet, Executable structural operational semantics in Maude, Journal of Logic and Algebraic Programming 67 (1-2) (2006) 226–293.

[45] A. Farzan, J. Meseguer, G. Roşu, Formal JVM code analysis in JavaFAN, in Proc. *AMAST'04*, Springer LNCS 3116, 132–147, 2004.

[46] A. Farzan, F. Cheng, J. Meseguer, G. Roşu, Formal analysis of Java programs in JavaFAN, in: Proc. CAV'04, Vol. 3114 of LNCS, 2004.

[47] E. B. Johnsen, O. Owe, E. W. Axelsen, A runtime environment for concurrent objects with asynchronous method calls, in: N. Martí-Oliet (Ed.), Proc. 5th. Intl. Workshop on Rewriting Logic and its Applications, Vol. 117, ENTCS, Elsevier, 2004.

[48] C. Braga, J. Meseguer, Modular rewriting semantics in practice, in: Proc. *WRLA'04*, Vol. 117, ENTCS, Elsevier, 2004, pp. 393–416.

[49] J. Meseguer, C. Braga, Modular rewriting semantics of programming languages, in Proc. AMAST'04, Springer LNCS 3116, 364–378, 2004.

[50] F. Chalub, C. Braga, A Modular Rewriting Semantics for CML, Journal of Universal Computer Science 10 (7) (2004) 789–807.

[51] F. Chalub, An implementation of Modular SOS in Maude, Master's thesis, Universidade Federal Fluminense, Niterói, RJ, Brazil (May 2005).

[52] A. Farzan, J. Meseguer, Partial order reduction for rewriting semantics of programming languages, in: G. Denker, C. Talcott (Eds.), Proc. 6th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS 176(4), Elsevier, 2007, pp. 61–78.

[53] M. Clavel, J. Santa-Cruz, ASIP + ITP: A verification tool based on algebraic semantics, in: F. J. López-Fraguas (Ed.), Actas de las V Jornadas sobre Programación y Lenguajes, PROLE 2005, Granada, España, Septiembre 14-16, 2005, Thomson, 2005, pp. 149–158.

[54] R. Sasse, Taclets vs. rewriting logic – relating semantics of Java, Master's thesis, Fakultät für Informatik, Universität Karlsruhe, Germany, technical Report in Computing Science No. 2005-16. (May 2005).
URL http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=ira/2005/16

[55] W. Ahrendt, A. Roth, R. Sasse, Automatic validation of transformation rules for java verification against a rewriting semantics., in: Proc. LPAR 2006, Vol. 3835 of LNCS, Springer-Verlag, 2005, pp. 412–426.

[56] M.-O. Stehr, C. L. Talcott, Practical techniques for language design and prototyping, in: J. L. Fiadeiro, U. Montanari, M. Wirsing (Eds.), Abstracts Collection of the Dagstuhl Seminar 05081 on Foundations of Global Computing. February 20 – 25, 2005. Schloss Dagstuhl, Wadern, Germany., 2005.

[57] M. d'Amorim, G. Roşu, An Equational Specification for the Scheme Language, Journal of Universal Computer Science 11 (7) (2005) 1327–1348, selected papers from the 9th Brazilian Symposium on Programming Languages (SBLP'05). Also Technical Report No. UIUCDCS-R-2005-2567, April 2005.

[58] R. Sasse, J. Meseguer, Java+itp: A verification tool based on hoare logic and algebraic semantics, in: G. Denker, C. Talcott (Eds.), Proc. 6th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS 176(4), Elsevier, 2007, pp. 29–46.

[59] M. Katelman, J. Meseguer, A rewriting semantics for abel with applications to hardware/software co-design and analysis, in: G. Denker, C. Talcott (Eds.), Proc. 6th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS 176(4), Elsevier, 2007, pp. 47–60.

[60] M. Hills, T. F. Şerbănuţă, G. Roşu, A rewrite framework for language definitions and for generation of efficient interpreters, in: Proc. of WRLA'06, Vol. 176(4) of ENTCS, Elsevier, 2007, pp. 215–231.

[61] A. Garrido, J. Meseguer, R. Johnson, Algebraic semantics of the C preprocessor and correctness of its refactorings, Tech. Rep. UIUCDCS-R-2006-2688, Department of Computer Science, University of Illinois at Urbana-Champaign (February 2006).
URL http://hdl.handle.net/2142/11162

[62] A. Farzan, Static and dynamic formal analysis of concurrent systems and languages: a semantics-based approach, Ph.D. thesis, University of Illinois at Urbana-Champaign (2007).

[63] M. AlTurki, J. Meseguer, Real-time rewriting semantics of Orc, in: M. Leuschel, A. Podelski (Eds.), Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative

Programming, PPDP 2007, Wroclaw, Poland, July 14-16, 2007, ACM, 2007, pp. 131–142.
URL `http://doi.acm.org/10.1145/1273920.1273938`

[64] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1) (1992) 73–155.

[65] R. Bruni, J. Meseguer, Semantic foundations for generalized rewrite theories., Theor. Comput. Sci. 360 (1-3) (2006) 386–414.

[66] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, C. Talcott, All About Maude – A High-Performance Logical Framework, Springer LNCS Vol. 4350, 2007.

[67] P. Viry, Equational rules for rewriting logic, Theoretical Computer Science 285 (2002) 487–517.

[68] J. Meseguer, K. Futatsugi, T. Winkler, Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents, in: Proceedings of the 1992 International Symposium on New Models for Software Architecture, Tokyo, Japan, November 1992, Research Institute of Software Engineering, 1992, pp. 61–106.

[69] N. Martí-Oliet, J. Meseguer, Rewriting logic as a logical and semantic framework, in: D. M. Gabbay, F. Guenthner (Eds.), Handbook of Philosophical Logic, Second Edition, Volume 9, Kluwer Academic Publishers, 2002, pp. 1–87.

[70] A. Verdejo, N. Martí-Oliet, Two case studies of semantics execution in Maude: CCS and LOTOS, Formal Methods in System Design 27 (1-2) (2005) 113–172.
URL `http://dx.doi.org/10.1007/s10703-005-2254-x`

[71] P. C. Ölveczky, J. Meseguer, Specification of real-time and hybrid systems in rewriting logic, Theoretical Computer Science 285 (2) (2002) 359–405.
URL `http://dx.doi.org/10.1016/S0304-3975(01)00363-2`

[72] P. C. Ölveczky, J. Meseguer, Semantics and pragmatics of Real-Time Maude, Higher-Order and Symbolic Computation 20 (1-2) (2007) 161–196.
URL `http://dx.doi.org/10.1007/s10990-007-9001-5`

[73] G. Agha, J. Meseguer, K. Sen, PMaude: Rewrite-based specification language for probabilistic object systems, Electr. Notes Theor. Comput. Sci. 153 (2) (2006) 213–239.

[74] J. Meseguer, A rewriting logic sampler, in: Proc. International Colloquium on Theoretical Aspects of Computing ICTAC05 (Hanoi, Vietnam, October 2005), Vol. 3722 of LNCS, Springer, 2005, pp. 1–28.

[75] J. Meseguer, R. Sharykin, Specification and analysis of distributed object-based stochastic hybrid systems, Tech. Rep. UIUCCDCS-R-2005-2649, University of Illinois at Urbana-Champaign, CS Department, to appear in *P*roc. Hybrid Systems 2006, Springer LNCS (October 2005).

[76] M. AlTurki, J. Meseguer, PVeStA: A parallel statistical model-checking and quantitative analysis tool, in Proc. CALCO 2011, Springer LNCS 6859, 386–392 (2011).

[77] M. Kim, M.-O. Stehr, C. L. Talcott, N. D. Dutt, N. Venkatasubramanian, A probabilistic formal analysis approach to cross layer optimization in distributed embedded systems, in: FMOODS 2007, Vol. 4468 of Lecture Notes in Computer Science, Springer, 2007, pp. 285–300.

[78] M. Katelman, J. Meseguer, J. C. Hou, Redesign of the lmst wireless sensor protocol through formal modeling and statistical model checking, in: Proc. FMOODS 2008, Vol. 5051 of LNCS, Springer, 2008, pp. 150–169.

[79] M. AlTurki, J. Meseguer, C. Gunter, Probabilistic modeling and analysis of DoS protection for the ASV protocol, Electr. Notes Theor. Comput. Sci. 234 (2009) 3–18.

[80] J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, M. Wirsing, Stable availability under denial of service attacks through formal patterns, in: J. de Lara, A. Zisman (Eds.), FASE, Vol. 7212 of Lecture Notes in Computer Science, Springer, 2012, pp. 78–93.

[81] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, Taming heterogeneity—the Ptolemy approach, Proceedings of the IEEE 91 (2) (2003) 127–144.

[82] G. Roşu, C. Ellison, W. Schulte, Matching logic: An alternative to Hoare/Floyd logic, in: Proc. AMAST'10, LNCS 6486, 2010, pp. 142–162.

[83] G. Roşu, A. Ştefănescu, Matching logic: A new program verification approach (nier track), in: Proc. ICSE'11, ACM, 2011.

[84] G. Rosu, A. Stefanescu, Towards a unified theory of operational and axiomatic semantics, in: Proc. ICALP'12, Vol. 7392 of LNCS, Springer, 2012, pp. 351–363.

[85] G. Rosu, A. Stefanescu, From hoare logic to matching logic reachability, in: Proc. FM'12, LNCS, Springer, 2012, to appear.

[86] G. Rosu, A. Stefanescu, Checking reachability using matching logic, in: Proc. OOPSLA'12, ACM, 2012, to appear.

[87] C. Braga, E. H. Haeusler, J. Meseguer, P. D. Mosses, Mapping modular SOS to rewriting logic, in: Proc. LOPSTR'02, LNCS 2664, 2002, pp. 262–277.

[88] F. Chalub, C. Braga, Maude MSOS tool, universidade Federal Fluminense, `www.ic.uff.br/~frosario/2o-workshop-vas-novembro-2004.pdf`.

[89] G. Roşu, T. F. Şerbănuţă, An overview of the K semantic framework, Journal of Logic and Algebraic Programming 79 (6) (2010) 397–434. `doi:10.1016/j.jlap.2010.03.012`.

[90] T. F. Serbanuta, G. Rosu, A trully concurrent semantics for the K framework based on graph transformations, in: Proc. ICGT'12, LNCS, 2012, to appear.

[91] T. F. Şerbănuţă, A rewriting approach to concurrent programming language design and semantics, Ph.D. thesis, University of Illinois at Urbana-Champaign, `https://www.ideals.illinois.edu/handle/2142/18252` (December 2010).

[92] G. Roşu, CS322, Fall 2003 - Programming Language Design: Lecture Notes, Tech. Rep. UIUCDCS-R-2003-2897, University of Illinois at Urbana-Champaign, Dept. of Computer Science, notes of a course taught at UIUC (2003).

[93] T. F. Şerbănuţă, G. Roşu, KRAM—extended report, Tech. Rep. http://hdl.handle.net/2142/17337, UIUC (September 2010).

[94] J. A. Goguen, G. Malcolm, Algebraic Semantics of Imperative Programs, Foundations of Computing, The MIT Press, 1996.
URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/026207172X`

[95] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe, Algebraic approaches to graph transformation: Basic concepts and double pushout approach, in: Handbook of graph grammars, Vol. 1, World Sci., 1997, pp. 163–246.

[96] J. Meseguer, Rewriting logic as a semantic framework for concurrency: a progress report, in: Proc. CONCUR'96, Pisa, August 1996, Springer LNCS 1119, 1996, pp. 331–372.

[97] C. Ellison, G. Roşu, An executable formal semantics of C with applications, in: Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12), ACM, 2012, pp. 533–544. `doi:10.1145/2103656.2103719`.

[98] P. Meredith, M. Hills, G. Roşu, A K Definition of Scheme, Tech. Rep. Department of Computer Science UIUCDCS-R-2007-2907, University of Illinois at Urbana-Champaign (2007).

[99] Y. Gurevich, J. K. Huggins, The semantics of the C programming language, in: Computer Science Logic, Vol. 702 of LNCS, 1993, pp. 274–308.

[100] J. V. Cook, E. L. Cohen, T. S. Redmond, A formal denotational semantics for C, Tech. Rep. 409D, Trusted Information Systems (September 1994).

[101] J. V. Cook, S. Subramanian, A formal semantics for C in Nqthm, Tech. Rep. 517D, Trusted Information Systems (Nov. 1994).

[102] M. Norrish, C formalised in HOL, Tech. Rep. UCAM-CL-TR-453, University of Cambridge (December 1998).

[103] N. S. Papaspyrou, Denotational semantics of ANSI C, Computer Standards and Interfaces 23 (3) (2001) 169–185.

[104] S. Blazy, X. Leroy, Mechanized semantics for the Clight subset of the C language, Journal of Automated Reasoning 43 (3) (2009) 263–288.

[105] N. S. Papaspyrou, A formal semantics for the C programming language, Ph.D. thesis, National Technical University of Athens (February 1998).

[106] M. AlTurki, D. Dhurjati, D. Yu, A. Chander, H. Inamura, Formal specification and analysis of timing properties in software systems, in: Proc. FASE, Vol. 5503 of LNCS, Springer, 2009, pp. 262–277.

[107] J. Misra, Computation orchestration: A basis for wide-area computing, in: M. Broy (Ed.), Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems Marktoberdorf, Germany, 2004, NATO ASI Series, 2004.

[108] J. Misra, W. R. Cook, Computation orchestration, Software and System Modeling 6 (1) (2007) 83–110.

[109] I. Wehrman, D. Kitchin, W. R. Cook, J. Misra, A timed semantics of Orc, Theor. Comput. Sci. 402 (2-3) (2008) 234–248.

[110] M. AlTurki, J. Meseguer, Reduction semantics and formal analysis of Orc programs, in: Proc. Workshop on Automated Specification and Verification of Web Systems (WWV'07), Vol. 200(3) of ENTCS, Elsevier, 2008, pp. 25–41.
URL http://dx.doi.org/10.1016/j.entcs.2008.04.091

[111] M. AlTurki, J. Meseguer, Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis, in: Proc. RTRTS'10, Vol. 36 of Electronic Proceedings in Theoretical Computer Science, CoRR, 2010, pp. 26–45.

[112] J. Bjørk, E. B. Johnsen, O. Owe, R. Schlatte, Lightweight time modeling in timed Creol, in: Proc. RTRTS'10, Vol. 36 of Electronic Proceedings in Theoretical Computer Science, CoRR, 2010, pp. 67–81.
URL http://dx.doi.org/10.4204/EPTCS.36.4

[113] J. Bjørk, F. de Boer, E. Johnsen, R. Schlatte, S. Tapia Tarifa, User-defined schedulers for real-time concurrent objects, Innovations in Systems and Software Engineering (To appear) 1–1510.1007/s11334-012-0184-5.
URL `http://dx.doi.org/10.1007/s11334-012-0184-5`

[114] M. Wirsing, A. Knapp, A formal approach to object-oriented software engineering, in: Proc. WRLA'96, Vol. 4 of ENTCS, 1996, pp. 322–360.
URL `http://dx.doi.org/10.1016/S1571-0661(04)00046-5`

[115] S. Nakajima, K. Futatsugi, An object-oriented modeling method for algebraic specifications in CafeOBJ, in: Proceedings of the 19th International Conference on Software Engineering, ICSE'97, Boston, Massachussets, May 17-23, 1997, ACM Press, 1997.
URL `http://dx.doi.org/10.1145/253228.253238`

[116] S. Nakajima, Using algebraic specification techniques in development of object-oriented frameworks, in: Proc. FM'99, Vol. 1709 of LNCS, Springer, 1999, pp. 1664–1683.
URL `http://dx.doi.org/10.1007/3-540-48118-4_38`

[117] J. L. Fernández Alemán, J. A. Toval Álvarez, Can intuition become rigorous? Foundations for UML model verification tools, in: Proc. ISSRE'00, IEEE, 2000, pp. 344–355.
URL `http://dx.doi.org/10.1109/ISSRE.2000.885885`

[118] A. Knapp, Generating rewrite theories from UML collaborations, in: K. Futatsugi, A. T. Nakagawa, T. Tamai (Eds.), Cafe: An Industrial-Strength Algebraic Formal Method, Elsevier, 2000, pp. 97–120.

[119] A. Knapp, A Formal Approach to Object-Oriented Software Engineering, Shaker Verlag, Aachen, Germany, 2001, phD thesis, Institut für Informatik, Universität München, 2000.

[120] M. Wirsing, A. Knapp, A formal approach to object-oriented software engineering, Theoretical Computer Science 285 (2) (2002) 519–560.
URL `http://dx.doi.org/10.1016/S0304-3975(01)00367-X`

[121] N. Aoumeur, G. Saake, Integrating and rapid-prototyping UML structural and behavioural diagrams using rewriting logic, in: Proc. CAiSE'02, Vol. 2348 of LNCS, Springer, 2002, pp. 296–310.
URL `http://dx.doi.org/10.1007/3-540-47961-9_22`

[122] M. Clavel, M. Egea, ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams, in: Proc. AMAST'06, Vol. 4019 of LNCS, Springer, 2006, pp. 368–373.
URL `http://dx.doi.org/10.1007/11784180_28`

[123] F. Mokhati, P. Gagnon, M. Badri, Verifying UML diagrams with model checking: A rewriting logic based approach, in: Proc. QSIC'07, IEEE,

2007, pp. 356–362.
URL http://dx.doi.org/10.1109/QSIC.2007.69

[124] F. Mokhati, M. Badri, Generating Maude specifications from UML use case diagrams, Journal of Object Technology 8 (2) (2009) 319–136.
URL http://www.jot.fm/issues/issue_2009_03/article2.pdf

[125] F. Mokhati, B. Sahraoui, S. Bouzaher, M. T. Kimour, A tool for specifying and validating agents' interaction protocols: From Agent UML to Maude, Journal of Object Technology 9 (3) (2010) 59–77.
URL http://www.jot.fm/contents/issue_2010_05/article2.html

[126] A. Boronat, J. A. Carsí, I. Ramos, Automatic reengineering in MDA using rewriting logic as transformation engine, in: Proc. CSMR'05, IEEE, 2005, pp. 228–231.
URL http://dx.doi.org/10.1109/CSMR.2005.14

[127] A. Boronat, MOMENT: A formal framework for MOdel ManageMENT, Ph.D. thesis, Universitat Politècnica de València, Spain (2007).

[128] A. Boronat, R. Heckel, J. Meseguer, Rewriting logic semantics and verification of model transformations, in: M. Chechik, M. Wirsing (Eds.), Proc. FASE'09, Vol. 5503 of LNCS, Springer, 2009, pp. 18–33.
URL http://dx.doi.org/10.1007/978-3-642-00593-0_2

[129] A. Boronat, J. Meseguer, MOMENT2: EMF model transformations in Maude, in: A. Vallecillo, G. Sagardui (Eds.), Actas de las XIV Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2009, San Sebastián, España, Septiembre 8-11, 2009, 2009, pp. 178–179.

[130] A. Boronat, J. Meseguer, An algebraic semantics for MOF, Formal Aspects of Computing 22 (3-4) (2010) 269–296.

[131] J. Meseguer, Membership algebra as a logical framework for equational specification, in: F. Parisi-Presicce (Ed.), Proc. WADT'97, Springer LNCS 1376, 1998, pp. 18–61.

[132] A. Boronat, J. Meseguer, Algebraic semantics of OCL-constrained metamodel specifications, in: Proc. TOOLS EUROPE'09, Vol. 33 of Lecture Notes in Business Information, Springer, 2009, pp. 96–115.
URL http://dx.doi.org/10.1007/978-3-642-02571-6_7

[133] A. Boronat, P. C. Ölveczky, Formal real-time model transformations in MOMENT2, in: Proc. FASE'10, Vol. 6013 of LNCS, Springer, 2010, pp. 29–43.
URL http://dx.doi.org/10.1007/978-3-642-12029-9_3

[134] E. A. Lee, Modeling concurrent real-time processes using discrete events, Ann. Software Eng. 7 (1999) 25–45.

[135] K. Bae, P. C. Ölveczky, T. H. Feng, S. Tripakis, Verifying Ptolemy II discrete-event models using Real-Time Maude, in: Proc. of ICFEM'09, Vol. 5885 of LNCS, Springer, 2009, pp. 717–736.
URL `http://dx.doi.org/10.1007/978-3-642-10373-5_37`

[136] K. Bae, P. C. Ölveczky, Extending the Real-Time Maude semantics of Ptolemy to hierarchical DE models, in: Proc. RTRTS'10, Vol. 36 of Electronic Proceedings in Theoretical Computer Science, CoRR, 2010, pp. 46–66.
URL `http://dx.doi.org/10.4204/EPTCS.36.3`

[137] P. C. Ölveczky, A. Boronat, J. Meseguer, Formal semantics and analysis of behavioral AADL models in Real-Time Maude, in: Proc. FMOODS'10, Vol. 6117 of LNCS, Springer, 2010, pp. 47–62.
URL `http://dx.doi.org/10.1007/978-3-642-13464-7_5`

[138] J. Meseguer, P. C. Ölveczky, Formalization and correctness of the PALS architectural pattern for real-time systems, in: 12th International Conference on Formal Engineering Methods (ICFEM 2010), Vol. 6447, Springer LNCS, 2010, pp. 303–320.

[139] K. Bae, P. C. Ölveczky, A. Al-Nayeem, J. Meseguer, Synchronous AADL and its formal analysis in Real-Time Maude, Tech. rep., University of Illinois at Urbana-Champaign, `http://hdl.handle.net/2142/25091` (2005).

[140] J. E. Rivera, F. Durán, A. Vallecillo, On the behavioral semantics of real-time domain specific visual languages, in: Proc. WRLA'10, Vol. 6381 of LNCS, Springer, 2010, pp. 174–190.
URL `http://dx.doi.org/10.1007/978-3-642-16310-4_12`

[141] K. Bae, P. C. Ölveczky, T. H. Feng, E. A. Lee, S. Tripakis, Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude, Science of Computer ProgrammingTo appear, `doi:10.1016/j.scico.2010.10.002`.

[142] K. Bae, P. C. Ölveczky, A. Al-Nayeem, J. Meseguer, Synchronous AADL and its formal analysis in Real-Time Maude, in: S. Qin, Z. Qiu (Eds.), Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings, Vol. 6991 of Lecture Notes in Computer Science, Springer, 2011, pp. 651–667.
URL `http://dx.doi.org/10.1007/978-3-642-24559-6_43`

[143] P. C. Ölveczky, Semantics, simulation, and formal analysis of modeling languages for embedded systems in Real-Time Maude, in: G. Agha, O. Danvy, J. Meseguer (Eds.), Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday, Vol. 7000 of Lecture Notes in Computer Science,

Springer, 2011, pp. 368–402.
URL `http://dx.doi.org/10.1007/978-3-642-24933-4_19`

[144] S. Miller, D. Cofer, L. Sha, J. Meseguer, A. Al-Nayeem, Implementing logical synchrony in integrated modular avionics, in: Proc. 28th Digital Avionics Systems Conference, IEEE, 2009.

[145] K. Bae, P. C. Ölveczky, A. Al-Nayeem, J. Meseguer, Synchronous AADL and its formal analysis in Real-Time Maude, Tech. rep., Department of Computer Science, University of Illinois at Urbana-Champaign, `http://hdl.handle.net/2142/25091` (2011).

[146] M. Katelman, J. Meseguer, S. Escobar, Directed-logical testing for functional verification of microprocessors, in: MEMOCODE'08, IEEE, 2008, pp. 89–100.
URL `http://dx.doi.org/10.1109/MEMCOD.2008.4547694`

[147] M. Katelman, J. Meseguer, `vlogsl`: A Strategy Language for Simulation-Based Verification of Hardware, in: S. Barner, I. Harris, D. Kroening, O. Raz (Eds.), Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference (HVC 2010), Vol. 6504 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2011, pp. 129 – 145.

[148] M. K. Katelman, A meta-language for functional verification, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (2011).

[149] P. Meredith, M. Katelman, J. Meseguer, G. Roşu, A formal executable semantics of Verilog, in: Proc. MEMOCODE'10, IEEE, 2010, pp. 179–188.
URL `http://dx.doi.org/10.1109/MEMCOD.2010.5558634`

[150] M. Katelman, S. Keller, J. Meseguer, Concurrent rewriting semantics and analysis of asynchronous digital circuits, in: Proc. WRLA'10, Vol. 6381 of LNCS, Springer, 2010, pp. 140–156.
URL `http://dx.doi.org/10.1007/978-3-642-16310-4_10`

[151] M. Hills, F. Chen, G. Roşu, Pluggable Policies for C, Tech. Rep. UIUCDCS-R-2008-2931, University of Illinois at Urbana-Champaign (2008).

[152] C. Ellison, T. F. Şerbănuţă, G. Roşu, A rewriting logic approach to type inference, in: Recent Trends in Algebraic Development Techniques, Vol. 5486 of LNCS, Springer, 2009, pp. 135–151.

[153] M. Alba-Castro, M. Alpuente, S. Escobar, Abstract certification of global non-interference in rewriting logic, in: Proc. FMCO, Vol. 6286 of LNCS, Springer, 2010, pp. 105–124.

[154] M. Alba-Castro, M. Alpuente, S. Escobar, Approximating non-interference and erasure in rewriting logic, in: Proc. SYNASC, IEEE, 2010, pp. 124–132.
URL `http://doi.ieeecomputersociety.org/10.1109/SYNASC.2010.25`

[155] J. Goguen, J. Meseguer, Security policies and security models, in: Proceedings of the 1982 Symposium on Security and Privacy, IEEE, 1982, pp. 11–20.

[156] H. Schorr, W. M. Waite, An efficient machine-independent procedure for garbage collection in various list structures, Commun. ACM 10 (8) (1967) 501–506.

[157] T. Hubert, C. Marché, A case study of C source code verification: the Schorr-Waite algorithm, in: SEFM, 2005, pp. 190–199.

[158] A. Loginov, T. W. Reps, M. Sagiv, Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm, in: SAS, 2006.