The Rewriting Logic Semantics Project

José Meseguer and Grigore Roşu

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

Abstract

Rewriting logic is a flexible and expressive logical framework that unifies denotational semantics and SOS in a novel way, avoiding their respective limitations and allowing very succinct semantic definitions. The fact that a rewrite theory's axioms include both equations and rewrite rules provides a very useful "abstraction knob" to find the right balance between abstraction and observability in semantic definitions. Such semantic definitions are directly executable as interpreters in a rewriting logic language such as Maude, whose generic formal tools can be used to endow those interpreters with powerful program analysis capabilities.

Keywords

Semantics of programming languages, rewriting logic, formal program analysis.

1 Introduction

The fact that rewriting logic specifications [36,9] provide an easy and expressive way to develop executable formal definitions of languages, which can then be subjected to different tool-supported formal analyses, is by now well established [63,6,64,59,57,38,61,14,51,62,26,24,34,7,40,41,12,11,25,19,52,1,58,21]. In fact, the just-mentioned papers by different authors are contributions to a collective ongoing research project which we call the *rewriting logic semantics project*. A first global snapshot of this project – emphasizing the fact that one can obtain quite efficient interpreters and program analysis tools from the semantic definitions essentially *for free* – was given in [41]. But this is a fast-moving area, so that new developments and the opportunity of discussing aspects less emphasized in [41] make it worthwhile for us to attempt giving here a second snapshot.

In our view, what makes this project promising is the combination of three interlocking facts:

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

(i) that, as explained in Sections 1.1 and 1.2, and further substantiated in the rest of this paper, rewriting logic is a flexible and expressive *logical framework* that unifies denotational semantics and SOS in a novel way, avoiding their respective limitations and allowing very succinct semantic definitions;

(ii) that rewriting logic semantic definitions are *directly executable* in a rewriting logic language such as Maude [16], and can thus become quite efficient interpreters; and

(iii) that *generic formal tools* such as the Maude LTL model checker [23], the Maude inductive theorem prover [17,18], and new tools under development such as a language-generic partial order reduction tool [25], allow us to amortize tool development cost across many programming languages, that can thus be endowed with powerful program analysis capabilities; furthermore, *genericity does not necessarily imply inefficiency*: in some cases the analyses so obtained outperform those of well-known language-specific tools [26,24].

1.1 Semantics: Equational vs. SOS

Two well-known semantic frameworks for programming languages are: equational semantics and structural operational semantics (SOS).

In *equational semantics*, formal definitions take the form of *semantic equations*, typically satisfying the *Church-Rosser* property. Both higher-order and first-order versions have been shown to be useful formalisms. There is a vast literature in these two areas that we do not attempt to survey. However, we can mention some early denotational semantics papers such as [54,55] and the surveys [53,44]. Similarly, we can mention [66,31,8] for early algebraic semantics papers, and [30] for a recent textbook.

We use the more neutral term *equational semantics* to emphasize the fact that denotational and algebraic semantics have many common features and can both be viewed as instances of a common equational framework. In fact, there isn't a rigid boundary between the two approaches, as illustrated, for example, by the conversion of higher-order semantic equations into first-order ones by means of explicit substitution calculi or combinators, the common use of initiality in both initial algebras and in solutions of domain equations, and a continuous version of algebraic semantics based on continuous algebras.

Strong points of equational semantics include:

- it has a *model-theoretic*, denotational semantics given by *domains* in the higher-order case, and by *initial algebras* in the first-order case;
- it has also a *proof-theoretic*, operational semantics given by *equational reduction* with the semantic equations;

- semantic definitions can be easily turned into efficient interpreters, thanks to efficient higher-order functional languages (ML, Haskell, etc.) and first-order equational languages (ACL2, OBJ, ASF+SDF, etc.);
- there is good higher-order and first-order theorem proving support.

However, equational semantics has the following drawbacks:

- it is well suited for *deterministic* languages such as conventional sequential languages or purely functional languages, but it is quite poorly suited to define the semantics of *concurrent languages*, unless the concurrency is that of a purely deterministic computation;
- one can *indirectly model*[1] some concurrency aspects with devices such as a scheduler, or lazy data structures, but a direct comprehensive modeling of all concurrency aspects remains elusive within an equational framework;
- semantic equations are typically *unmodular*, i.e., adding new features to a language often requires *extensive redefinition* of earlier semantic equations.

In SOS formal definitions take the form of *semantic rules*. SOS is a proof-theoretic approach, focusing on giving a detailed step-by-step formal description of a program's execution. The semantic rules are used as inference rules to reason about what computation steps are possible. Typically, the rules follow the syntactic structure of programs, defining the semantics of a language construct in terms of that of its parts. The *locus classicus* is Plotkin's Aarhus lectures [49]; there is again a vast literature on the topic that we do not attempt to survey; for a good textbook introduction see [33].

Strong points of SOS include:

- it is a general, yet quite intuitive formalism, allowing detailed *step-by-step* modeling of program execution;
- it has a simple *proof-theoretic* semantics using semantic rules as inference rules;
- it is fairly well suited to model *concurrent languages*, and can also deal well with the detailed execution of deterministic languages;
- it allows *mathematical reasoning and proof*, by reasoning inductively or coinductively about the inference steps.

However, SOS has the following drawbacks:

- although specific proposals have been made for *categorical models* for certain SOS formats, such as, for example, Turi's functorial SOS [60] and Gadducci

[1] Two good examples of indirectly modeling concurrency within a purely functional framework are the ACL2 semantics of the JVM using a scheduler [43], and the use of lazy data structures in Haskell to analyze cryptographic protocols [2].

and Montanari's tile models [28], it seems however fair to say that, so far, SOS has no commonly agreed upon model-theoretic semantics;

- in its standard formulation it imposes a centralized *interleaving semantics* of concurrent computations, which may be unnatural in some cases (for example for highly decentralized and asynchronous mobile computations); this problem is avoided in "reduction semantics," which is different from SOS and is in fact a special case of rewriting semantics (see Section 2.5);
- standard SOS definitions are notoriously *unmodular*, unless one adopts Mosses' MSOS framework (see Section 4.4);
- although some tools have been built to execute SOS definitions (see for example [20,32,48]), tool support for verifying properties is perhaps less developed than for equational semantics.

1.2 Unifying SOS and Equational Semantics: the Abstraction Knob

For the most part, equational semantics and SOS have lived separate lives. Pragmatic considerations and differences in taste tend to dictate which framework is adopted in each particular case. For concurrent languages SOS is clearly superior and tends to prevail as the formalism of choice, but for deterministic languages equational approaches are also widely used. Of course there are also practical considerations of tool support for both execution and formal reasoning.

In the end, equational semantics and SOS, although each very valuable in its own way, are "single hammer" approaches. Would it be possible to seamlessly *unify* them within a more flexible and general framework? Could their respective limitations be overcome when they are thus unified? Our proposal is that rewriting logic [36,9] does indeed provide one such unifying framework. The key to this, indeed very simple, unification is what we call rewriting logic's *abstraction knob*. The point is that in equational semantics' model-theoretic approach entities are *identified by the semantic equations*, and have unique *abstract denotations* in the corresponding models. In our knob metaphor this means that in equational semantics the abstraction knob is *always turned all the way up to its maximum position*. By contrast, one of the key features of SOS is providing a very detailed, step-by-step formal description of a language's evaluation mechanisms. As a consequence, most entities – except perhaps for built-in data, stores, and environments, which are typically treated on the side – are *primarily syntactic*, and computations are described in full detail. In our metaphor this means that in SOS the abstraction knob is *always turned down to its minimum position*.

How is the unification and corresponding availability of an abstraction knob achieved? Roughly speaking,[2] a rewrite theory is a triple (Σ, E, R),

[2] We postpone discussion of "equational reduction strategies" μ, and "frozen" argument

with (Σ, E) an equational theory with Σ a signature of operations and sorts, and E a set of (possibly conditional) equations, and with R a set of (possibly conditional) rewrite rules. Equational semantics is obtained as the special case in which $R = \varnothing$, so we only have the semantic equations E and the abstraction knob is turned up to its maximum position. Roughly speaking,[3] SOS (with unlabeled transitions) is obtained as the special case in which $E = \varnothing$, and we only have (possibly conditional) rules R rewriting purely syntactic entities (terms), so that the abstraction knob is turned down to the minimum position.

Rewriting logic's "abstraction knob" is precisely its crucial distinction between equations E and rules R in a rewrite theory (Σ, E, R). *States of the computation* are then E-equivalence classes, that is, *abstract elements* in the initial algebra $T_{\Sigma/E}$. Because of rewriting logic's "Equality" inference rule (see Section 2.2) a rewrite with a rule in R is understood as a transition $[t] \longrightarrow [t']$ between such abstract states. The knob, however, can be turned up or down. We can turn it *all the way down to its minimum* by converting all equations into rules, transforming (Σ, E, R) into $(\Sigma, \varnothing, R \cup E)$. This gives us the most concrete, SOS-like semantic description possible. Can we turn the knob "all the way up," in the sense of converting all rules into equations? Only if the system we are describing is *deterministic* (for example, the semantic definition of a sequential language) is this a sound procedure. In that case, the equational theory $(\Sigma, R \cup E)$ should be Church-Rosser, and we do indeed obtain a most-abstract-possible, purely equational semantics out of the less abstract specification (Σ, E, R), or even out of the most concrete possible specification $(\Sigma, \varnothing, R \cup E)$.

What can we do in general to make a specification *as abstract as possible*? We can identify a subset $R_0 \subseteq R$ such that: (1) $R_0 \cup E$ is Church-Rosser; and (2) R_0 is biggest possible with this property. In actual language specification practice this is not hard to do. We illustrate this idea with a simple example language in Section 3.1. Essentially, we can use semantic equations for most of the sequential features of a programming language: only when interactions with memory could lead to nondeterminism (particularly if the language has threads, or they could later be added to the language in an extension) or for intrinsically concurrent features are rules (as opposed to equations) really needed. In our experience, it is often possible to specify most of the semantic axioms with equations, with relatively few rules needed for truly concurrent or nondeterministic features. For example, the semantics of the JVM described in [26,24] has about 300 equations and 40 rules; and that of Java described in [24] has about 600 equations but only 15 rules. A semantics for an ML-like

information ϕ to Sections 2.1 and 2.2. In more detail, a rewrite theory will be axiomatized as a tuple $(\Sigma, E, \mu, R, \phi)$.

[3] We gloss over the technical difference that in SOS all computations are "one-step" computations, even if the step is a big one, whereas in rewriting logic, because of its built-in "Transitivity" inference rule (see Section 2.2) the rewriting relation is always transitive; we give a more detailed comparison in Section 2.5.

language with threads given in [41] has only two rules.

This distinction between equations and rules, besides giving to equational semantics and SOS their due in a way not possible for the other alternative if we were to remain within each of these formalisms, has also important practical consequences for *program analysis*; because it affords a massive *state space reduction* which can make formal analyses such as breadth-first search and model checking enormously more efficient. Because of state-space explosion, such analyses could easily become infeasible if we were to use an SOS-like specification in which all computation steps are described with rules. This capacity of dealing with abstract states is a crucial reason why our generic tools, when instantiated to a given programming language definition, tend to result in program analysis tools of competitive performance. Of course, the price to pay in exchange for abstraction is a *coarser level of granularity* in respect to what aspects of a computation are *observable* at that abstraction level. For example, when analyzing a sequential program using a semantics in which most sequential features have been specified with equations, all sequential subcomputations will be abstracted away, and the analysis will focus on memory and thread interactions. If a finer analysis is needed, we can often obtain it by "turning down the abstraction knob" to the right observability level by *converting some equations into rules*. That is, we can regulate the knob to find for each kind of analysis the best possible balance between abstraction and observability.

1.3 About this Paper

The rest of the paper is organized as follows. Background on membership equational logic and rewriting logic is given in Sections 2.1–2.4. The relationship to denotational semantics and SOS is discussed in greater detail in Section 2.5. We then discuss different language specification styles possible within the common rewriting logic framework in Section 3, illustrate one of these styles by the specification of a simple programming language in Section 3.1, and summarize other language specification case studies in Section 3.2. Program analysis techniques and tools are discussed in Section 4, including search and model checking analyses (4.1), abstract-semantics-based analyses (4.2), logics of programs and semantics-based theorem proving (4.3), and modularity and the MSOS tool (4.4). We end with some concluding remarks in Section 5.

2 Rewriting Logic Semantics

2.1 Membership Equational Logic

A membership equational logic (MEL) [37] *signature* is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ a many-kinded

signature, and $S = \{S_k\}_{k \in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A MEL Σ-algebra A contains a set A_k for each kind $k \in K$, a function $A_f \colon A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1 \cdots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, with the meaning that the elements in sorts are well-defined, while elements without a sort are *errors*. We write $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ to denote, respectively, the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of kinded variables. Given a MEL signature Σ, *atomic formulae* have either the form $t = t'$ (Σ-equation) or $t : s$ (Σ-membership) with $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$; and Σ-*sentences* are conditional formulae of the form "$(\forall X)\ \varphi$ if $\bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$", where φ is either a Σ-equation or a Σ-membership, and all the variables in φ, p_i, q_i, and w_j are in X. A MEL theory is a pair (Σ, E) with Σ a MEL signature and E a set of Σ-sentences. We refer to [37] for the detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, and initial and free algebras. In particular, given a MEL theory (Σ, E), its initial algebra is denoted $T_{\Sigma/E}$; its elements are E-equivalence classes of ground terms in T_Σ. Order-sorted notation $s_1 < s_2$ can be used to abbreviate the conditional membership "$(\forall x : k)\ x : s_2$ if $x : s_1$". Similarly, an operator declaration $f \colon s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom "$(\forall x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s$ if $\bigwedge_{1 \leq i \leq n} x_i : s_i$". We write $(\forall x_1 : s_1, \ldots, x_n : s_n)\ t = t'$ in place of "$(\forall x_1 : k_1, \ldots, x_n : k_n)\ t = t'$ if $\bigwedge_{1 \leq i \leq n} x_i : s_i$".

For execution purposes we typically impose some requirements on a MEL theory. First of all, its sentences may be decomposed as a union $E \cup A$, with A a set of equations that we will reason *modulo* (for example, A may include associativity, commutativity and/or identity axioms for some of the operators in Σ). Second, the sentences E are typically required to be Church-Rosser [4] modulo A, so that we can use the conditional equations E as equational rewrite rules modulo A. Third, for some applications it is useful to make the equational rewriting relation [5] *context-sensitive*. This can be accomplished by specifying a function $\mu : \Sigma \longrightarrow \mathbb{N}^*$ assigning to each function symbol $f \in \Sigma$ (with, say, n arguments) a list $\mu(f) = i_1 \ldots i_k$ of *argument positions*, with $1 \leq i_j \leq n$, which must be fully evaluated (up to the context-sensitive equational reduction stategy specified by μ) in the order specified by the list $i_1 \ldots i_k$ before applying any equations whose lefthand sides have f as their top symbol. For example, for $f = \mathit{if\_then\_else\_fi}$ we may give $\mu(f) = \{1\}$, meaning that the first argument must be fully evaluated before the equations for $\mathit{if\_then\_else\_fi}$

[4] See [5] for a detailed study of equational rewriting concepts and proof techniques for MEL theories.

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

are applied[6]. Therefore, for execution purposes we can specify a MEL theory as a triple $(\Sigma, E \cup A, \mu)$, with A the axioms we rewrite modulo, and with μ the map specifying the context-sensitive equational reduction strategy.

2.2 Rewrite Theories

A *rewriting logic specification or theory* is a tuple $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$, with:

- $(\Sigma, E \cup A, \mu)$ a MEL theory with "modulo" axioms A and context-sensitive equational reduction strategy μ.
- R a set of *labeled conditional rewrite rules* of the general form

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

  where the variables appearing in all terms are among those in X, terms in each rewrite or equation have the same kind, and in each membership $v_j : s_j$ the term v_j has kind $[s_j]$; and
- $\phi : \Sigma \longrightarrow \mathcal{P}(\mathbb{N})$ a mapping assigning to each function symbol $f \in \Sigma$ (with, say, n arguments) a set $\phi(f) = \{i_1, \ldots, i_k\}$, $1 \leq i_1 < \ldots < i_k \leq n$ of *frozen argument positions*[7] under which it is forbidden to perform any rewrites.

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

The frozenness information is important in practice to forbid certain rewritings. For example, when defining the rewriting semantics of a process calculus, one may wish to require that in prefix expressions $\alpha.P$ the operator $\_.\_$ is *frozen in the second argument*, that is, $\phi(\_.\_) = \{2\}$, so that P cannot be rewritten under a prefix. The frozenness idea can be extended to variables in terms as follows: given a Σ-term $t \in T_\Sigma(X)$, we call a variable $x \in \mathit{vars}(t)$ *frozen* in t iff there is a nonvariable position $\alpha \in \mathbb{N}^*$ such that $t/\alpha = f(u_1, \ldots, u_i, \ldots, u_n)$, with $i \in \phi(f)$, and $x \in \mathit{vars}(u_i)$. Otherwise, we call $x \in X$ *unfrozen*. Similarly, given Σ-terms $t, t' \in T_\Sigma(X)$, we call a variable $x \in X$ *unfrozen* in t and t' iff it is unfrozen in both t and t'.

[6] Maude has a functional sublanguage whose modules are membership equational theories. Maps μ specifying context-sensitive equational reduction strategies are called *evaluation strategies* [16], and $\mu(f) = i_1 \ldots i_k$ is specified with the `strat` keyword followed by the string $(i_1\ \ldots\ i_k\ 0)$, with the last 0 indicating evaluation at the top of the function symbol f.

[7] In Maude, $\phi(f) = \{i_1, \ldots, i_k\}$ is specified by declaring f with the `frozen` attribute, followed by the string $(i_1\ \ldots\ i_k)$.

Note that a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, \phi, R)$ specifies two kinds of *context-sensitive* rewriting requirements: (1) equational rewriting with E modulo A is made context-sensitive by μ; and (2) nonequational rewriting with R is made context-sensitive by ϕ. But the maps μ and ϕ impose *different types* of context-sensitive requirements: (1) $\mu(f)$ specifies a list of arguments that *must be* fully evaluated with the equations E (up to the strategy μ) before equations for f are applied; and (2) $\phi(f)$ specifies arguments that *must never be* rewritten with the rules R under the operator f. The maps μ and ϕ substantially increase the expressive power of rewriting logic for semantic definition purposes, because various order-of-evaluation and context-sensitive information, which would have to be specified by explicit rules in a formalism like *SOS*, becomes implicit and is encapsulated in μ and ϕ.

2.3 Rewriting Logic Deduction

Given $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$, the sentences that $\mathcal{R}$ proves are universally quantified rewrites of the form $(\forall X)\ t \longrightarrow t'$, with $t, t' \in T_\Sigma(X)_k$, for some kind k, which are obtained by finite application of the following *rules of deduction*:

- **Reflexivity**. For each $t \in T_\Sigma(X)$, $\dfrac{}{(\forall X)\ t \longrightarrow t}$
- **Equality**. $\dfrac{(\forall X)\ u \longrightarrow v \quad E \cup A \vdash (\forall X) u = u' \quad E \cup A \vdash (\forall X) v = v'}{(\forall X)\ u' \longrightarrow v'}$
- **Congruence**. For each $f : k_1 \dots k_n \longrightarrow k$ in Σ, with $\{1, \dots, n\} - \phi(f) = \{j_1, \dots, j_m\}$, with $t_i \in T_\Sigma(X)_{k_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{k_{j_l}}$, $1 \leq l \leq m$,

$$\frac{(\forall X)\ t_{j_1} \longrightarrow t'_{j_1} \quad \dots \quad (\forall X)\ t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\ f(t_1, \dots, t_{j_1}, \dots, t_{j_m}, \dots, t_n) \longrightarrow f(t_1, \dots, t'_{j_1}, \dots, t'_{j_m}, \dots, t_n)}$$

- **Replacement**. For each $\theta : X \longrightarrow T_\Sigma(Y)$ with, say, $X = \{x_1, \dots, x_n\}$, and $\theta(x_l) = p_l$, $1 \leq l \leq n$, and for each rule in R of the form,

$$q : (\forall X)\ t \longrightarrow t' \ \text{ if } \ (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \longrightarrow w'_k)$$

with $Z = \{x_{j_1}, \dots, x_{j_m}\}$ the set of unfrozen variables in t and t', then,

$$(\bigwedge_r (\forall Y)\ p_{j_r} \longrightarrow p'_{j_r})$$

$$\frac{(\bigwedge_i (\forall Y)\ \theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j (\forall Y)\ \theta(v_j) : s_j) \wedge (\bigwedge_k (\forall Y)\ \theta(w_k) \longrightarrow \theta(w'_k))}{(\forall Y)\ \theta(t) \longrightarrow \theta'(t')}$$

where for $x \in X - Z$, $\theta'(x) = \theta(x)$, and for $x_{j_r} \in Z$, $\theta'(x_{j_r}) = p'_{j_r}$, $1 \leq r \leq m$.

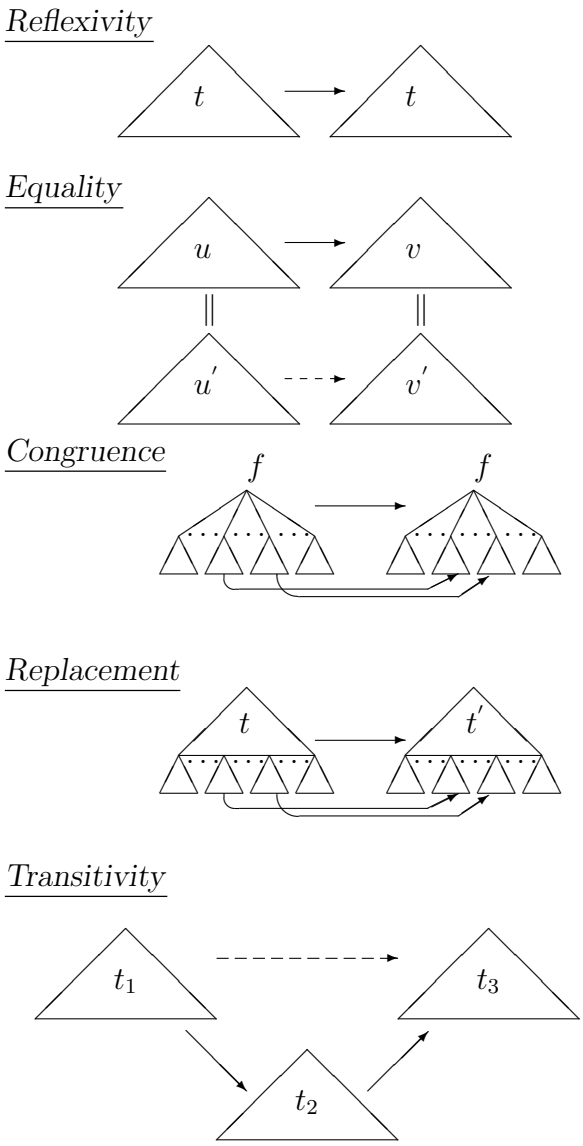

Fig. 1. Visual representation of rewriting logic deduction.

- **Transitivity**

$$\frac{(\forall X)\ t_1 \longrightarrow t_2 \qquad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$$

We can visualize the above inference rules as in Figure 1.

The notation $\mathcal{R} \vdash t \longrightarrow t'$ states that the sequent $t \longrightarrow t'$ is *provable* in the theory $\mathcal{R}$ using the above inference rules. Intuitively, we should think of the inference rules as different ways of *constructing* all the (finitary) *concurrent computations* of the concurrent system specified by $\mathcal{R}$. The "Reflexivity" rule says that for any state t there is an *idle transition* in which nothing changes. The "Equality" rule specifies that the states are in fact equivalence

classes modulo the equations E. The "Congruence" rule is a very general form of "sideways parallelism," so that each operator f can be seen as a *parallel state constructor*, allowing its nonfrozen arguments to evolve in parallel. The "Replacement" rule supports a different form of parallelism, which could be called "parallelism under one's feet," since besides rewriting an instance of a rule's lefthand side to the corresponding righthand side instance, the state fragments in the substitution of the rule's variables can also be rewritten, provided the variables involved are not frozen. Finally, the "Transitivity" rule allows us to build longer concurrent computations by composing them sequentially.

For execution purposes a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ should satisfy some basic requirements. These requirements are assumed to hold by a rewriting logic language such as Maude. First, in the MEL theory $(\Sigma, E \cup A, \mu)$ E should be ground Church-Rosser modulo A – for A a set of equational axioms for which matching modulo A is decidable – and ground terminating modulo A up to the context-sensitive strategy μ [8]. Second, the rules R should be *coherent* with E modulo A [65]; intuitively, this means that, to get the effect of rewriting in equivalence classes modulo $E \cup A$, we can always first simplify a term with the equations E to its canonical form modulo A, and then rewrite with a rule in R. Finally, the rules in R should be *admissible* [16], meaning that in a rule of the form (1) on page 8, besides the variables appearing in t there can be extra variables in t', provided that they also appear in the condition and that they can all be *incrementally instantiated* by either matching a pattern in a "matching equation" or performing breadth first search in a rewrite condition (see [16] for a detailed description of admissible equations and rules).

2.4 Operational and Denotational Semantics of Rewrite Theories

A rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ has both a *deduction-based operational semantics*, and an *initial model denotational semantics*. Both semantics are defined naturally out of the proof theory described in Section 2.3. The deduction-based operational semantics of $\mathcal{R}$ is defined as the collection of *proof terms* [36,9] 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

[8] The μ-termination condition may be dropped for programming language specifications in which some equationally defined language constructs may not terminate. Even the ground Church-Rosser property modulo A may be relaxed, by restricting it to terms in some "observable kinds" of interest. The point is that there may be some "unobservable" kinds for which several different but semantically equivalent terms can be derived by equational simplification: all we need in practice is that the operations are ground Church-Rosser modulo A for terms in an observable kind, such as that of values, so that a unique canonical form is then reached for them if it exists.

Section 2.3. As already mentioned, all such proof trees describe all the possible *finitary concurrent computations* of the concurrent system axiomatized by $\mathcal{R}$. When we specify $\mathcal{R}$ as a Maude module and rewrite a term t with the `rewrite` or `frewrite` commands, obtaining a term t' as a result, we can use Maude's `trace` mode to obtain what amounts to a proof term $\alpha : t \longrightarrow t'$ of the particular rewrite proof built by the Maude interpreter.

A rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ has also a model theory. The models of $\mathcal{R}$ are *categories* with a $(\Sigma, E \cup A)$-algebra structure [36,9]. 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? It turns out that the class of models of a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \mu, R, \phi)$ has an *initial model* $\mathcal{T}_{\mathcal{R}}$ [36,9]. 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. Of course, α and β should have identical beginning states and identical ending states. By the "Equality" rule this forces $E \cup A \vdash t = u$, and $E \cup A \vdash t' = u'$. That, is, the category objects in $\mathcal{T}_{\mathcal{R}}$ are $E \cup A$-equivalence classes $[t]$ of ground Σ-terms, which denote the states of our system. The arrows or morphisms in $\mathcal{T}_{\mathcal{R}}$ are *equivalence classes of proof terms*, so that $[\alpha] = [\beta]$ iff both proof terms denote the same concurrent computation according to the "true concurrency" axioms. Such axioms are very natural. They for example express that the "Transitivity" rule behaves as an arrow composition, and is therefore associative. Similarly, the "Reflexivity" rules provides an identity arrow for each object, satisfying the usual identity laws.

2.5 Rewriting Logic Semantics of Programming Languages

Rewriting logic's operational and denotational semantics apply in particular to the specification of programming languages. We define the semantics of a (possibly concurrent) programming language, say $\mathcal{L}$, by specifying a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, (E \cup A)_{\mathcal{L}}, \mu_{\mathcal{L}}, R_{\mathcal{L}}, \phi_{\mathcal{L}})$, where $\Sigma_{\mathcal{L}}$ specifies $\mathcal{L}$'s *syntax* and the auxiliary operators (store, environment, etc.), $(E \cup A)_{\mathcal{L}}$ specifies the semantics of all the *deterministic features* of $\mathcal{L}$ and of the auxiliary semantic operations, the rewrite rules $R_{\mathcal{L}}$ specify the semantics of all the *concurrent features* of $\mathcal{L}$, and $\mu_{\mathcal{L}}$ and $\phi_{\mathcal{L}}$ specify additional context-sensitive rewriting requirements for the equations $(E \cup A)_{\mathcal{L}}$ and the rules $R_{\mathcal{L}}$. Section 3.1 gives a detailed case study of a rewriting semantics $\mathcal{R}_{\mathcal{L}}$ for $\mathcal{L}$ a simple programming language.

The relationships with equational semantics and SOS can now be described more precisely. First of all, note that when $R = \varnothing$, the only possible arrows are identities, so that the initial model $\mathcal{T}_{\mathcal{R}}$ becomes isomorphic to the initial algebra $T_{\Sigma/E \cup A}$. That is, traditional initial algebra semantics [29], which pro-

vides the models for algebraic denotational semantics, appears as a special case of rewriting logic's initial model semantics.

As already mentioned, we can also obtain SOS as the special case in which we "turn the abstraction knob" all the way down to the minimum position by turning all equations into rules. Intuitively, an SOS rule of the form

$$\frac{P_1 \longrightarrow P_1' \quad \ldots \quad P_n \longrightarrow P_n'}{Q \longrightarrow Q'}$$

corresponds to a rewrite rule with *rewrites in its condition*

$$Q \longrightarrow Q' \;\; \textit{if} \;\; P_1 \longrightarrow P_1' \wedge \ldots \wedge P_n \longrightarrow P_n'$$

There are however some technical differences between the meaning of a transition $P \longrightarrow Q$ in SOS and a sequent $P \longrightarrow Q$ in rewriting logic. In SOS a transition $P \longrightarrow Q$ is always a *one-step* transition. Instead, because of "Reflexivity" and "Transitivity", a rewriting logic sequent $P \longrightarrow Q$ may involve many rewrite steps; furthermore, because of "Congruence", such steps may correspond to rewriting subterms. These technical differences present no real difficulty for expressing SOS within rewriting logic: as shown in detail in [41], we can just "dumb down" the rewriting logic inference to force one-step rewrites in conditions. This can be easily accomplished by adding two auxiliary operators $[\_]$ and $\langle\_\rangle$, so that SOS rules of the form above can be exactly simulated by conditional rewrite rules of the form

$$[Q] \longrightarrow \langle Q' \rangle \;\; \textit{if} \;\; [P_1] \longrightarrow \langle P_1' \rangle \wedge \ldots \wedge [P_n] \longrightarrow \langle P_n' \rangle$$

In general, SOS rules may have *labels*, *decorations*, and *side conditions*. In fact, there are many SOS rule variants and formats. For example, additional semantic information about stores or environments can be used to decorate an SOS rule. Therefore, showing in detail how SOS rules in each particular variant or format can be faithfully represented by corresponding rewrite rules would be a tedious business. Fortunately, Peter Mosses, in his modular structural operational semantics (MSOS) [45,46,47], has managed to neatly pack all the various pieces of semantic information usually *scattered throughout* a standard SOS rule *inside labels on transitions*, where now labels have a record structure whose fields correspond to the different semantic components (the store, the environment, action traces for processes, and so on) *before and after* the transition thus labeled is taken. The paper [40] defines a faithful representation of an MSOS specification $\mathcal{S}$ as a corresponding rewrite theory $\tau(\mathcal{S})$, provided that the MSOS rules in $\mathcal{S}$ are in a suitable normal form.

A different approach, also subsumed by rewriting logic semantics, is sometimes described as *reduction semantics*. It goes back to Berry and Boudol's

Chemical Abstract Machine (Cham) [4], and has been used to give semantics to different concurrent calculi and programming languages (see [4,42] for two early references). Since the 1990 San Miniato Workshop on Concurrency, where both the Cham and rewriting logic were presented [22], it has been clearly understood that these are two closely related formalisms, so that each Cham can be naturally seen as a rewrite theory (see [36] Section 5.3.3, and [4]). In essence, a reduction semantics, either of the Cham type or with a different choice of basic primitives, can be naturally seen as a special type of rewrite theory $\mathcal{R} = (\Sigma, A, R, \phi)$, where A consists of *structural axioms*, e.g., associativity and commutativity of multiset union for the Cham, and R is a set of *unconditional* rewrite rules. The frozenness information ϕ is specified by giving explicit inference rules, stating which kind of *congruence* is permitted for each operator for rewriting purposes. This last point illustrates why rewriting logic semantic definitions can be *more succinct* than SOS or reduction semantics definitions, because rewriting logic's "Congruence" rule, together with the frozenness information ϕ, implicitly takes care of context-sensitive information that has to be handled by explicit rules in both SOS and reduction semantics definitions.

Evaluation context semantics [27] is a variant of reduction semantics in which the applicability of reductions is controlled by requiring them to occur in definable *evaluation contexts*. In rewriting logic one can obtain the same effect again by making use of the frozenness information. However, the rewriting logic specification style is slightly different, because operations are suppossed congruent by dafault; one needs to explicitely state which operations are *not* congruent (or frozen).

3 Specifying Programming Languages

In computer science there are typically many different ways to *implement* a given problem or system, each with its own advantages and disadvantages. Similarly, there can be many different styles to *specify* the same system or design in rewriting logic, depending upon one's goals, such as operational efficiency, verification of properties, mathematical clarity, modularity, or just one's personal taste. It is therefore not surprising that different, semantically equivalent rewriting logic definitional styles are possible for specifying a given programming language $\mathcal{L}$.

In what follows we briefly discuss three definitional styles that we have investigated, together with their advantages and disadvantages. A fourth definitional style, providing a fully modular specification methodology, is described in detail in [7,40], and is compared to the MSOS methodology in Section 4.4. Yet another definitional style, called *reduction-context semantics* and yielding a very direct connection between a partially executed program and the machine state is proposed and illustrated in [58]. What is common to all

these styles is the fact that there is a sort `State` together with appropriate constructors to store state information necessary to define the various language constructs, such as locations, values, environments, stores, etc., as well as means to define the two important semantic aspects of each language construct, namely: (1) the value it evaluates to in a given state; and (2) the state resulting after its evaluation. We explain the three definitional styles informally by means of two sample language constructs: a binary addition operation and a conditional statement. All three language definitional styles assume an expression-based syntax, that is, all language constructs generate plain expressions (as opposed to generating arithmetic expressions, boolean expressions, statements, etc.). We postpone the description of the state infrastructure until Section 3.1.

Separating Evaluation and State Update

Following a divide-and-concur philosophy, an intuitive definitional style for a language is one focusing on evaluation and on state update as two different operations. To achieve this, one needs to define two operations [9] as follows:

```
op eval  : Exp State -> Value .
op state : Exp State -> State .
```

The first operation takes an expression in a state and evaluates it to a value, while the second calculates the state obtained after evaluation. The semantics of addition can then be defined as follows:

```
eq eval(E1 + E2, S) = add(eval(E1, S), eval(E2, state(E1, S))) .
eq add(int(I1), int(I2)) = int(I1 + I2) .

eq state(E1 + E2, S) = state(E2, state(E1, S)) .
```

The above equations reflect the fact that the expressions `E1` and `E2` are evaluated in a left-to-right order. Indeed, `E2` is evaluated in the state obtained after the evaluation of `E1`. The evaluation order of subexpressions is very important, because in our language the evaluation of expressions can have side effects; so different evaluation orders potentially lead to different results. To keep the various "buil-in" theories/modules conceptually disconnected, the

[9] In all our examples throughout the paper we use Maude syntax [16], which is so close to the corresponding mathematical notation for defining rewrite theories as to be almost self-explanatory. The general point to keep in mind is that each item: a sort, a subsort, an operation, an equation, a rule, etc., is declared with an obvious keyword: `sort`, `subsort`, `op`, `eq` (or `ceq` for conditional equations), `rl` (or `crl` for conditional rules), etc., with each declaration ended by a space and a period. Another important point is the use of "mixfix" user-definable syntax, with the argument positions specified by underbars; for example: `if_then_else_fi`. As illustrated in Section 3.1, this is very useful for programming language definitions, since a language's concrete syntax (except perhaps for minor lexical details) can be preserved in the Maude specification.

different types of values (integers, booleans, etc.) are wrapped with appropriate constructors – as oppossed to declaring them all subsorts of a generic expression supersort. Therefore, one needs to first "unwrap" them in order to apply basic operations. For example, in the above we used an auxiliary (semantic) binary operation `add` defined on (integer) values as expected.

The semantics of the conditional construct can be defined in a similar style:

```
eq eval(if E then E1 else E2, S)
 = if true?(eval(E, S))
   then eval(E1, state(E, S))
   else eval(E2, state(E, S))
   fi
eq true?(bool(true)) = true .
eq true?(bool(false)) = false .

eq state(if E then E1 else E2, S)
 = if true?(eval(E, S))
   then state(E1, state(E, S))
   else state(E2, state(E, S))
   fi
```

Note that a test operation, `true?`, was defined on boolean values to "unwrap" their truth value in the builtin (Maude) bool module. This could have been avoided if we used conditional equations with matching in conditions. It is becoming increasingly clear that code repetition is a major drawback of this definitional style. This motivates the following, more compact definitional style.

Merging Evaluation and State Update

By pairing values and states, one can define evaluation and state update together. Consider a new sort `ValueStatePair` and the constructor operation for it:

```
op <_,_> : Value State -> ValueStatePair .
```

We can then define an operation

```
op eval : Exp State -> ValueStatePair .
```

on such pairs as follows:

```
ceq eval(E1 + E2, S) = < int(I1 + I2), S2 >
   if < int(I1), S1 > := eval(E1, S)
   /\ < int(I2), S2 > := eval(E2, S1) .

ceq eval(if E then E1 else E2, S) =
  = eval(if B then E1 else E2 fi, Sb)
   if < bool(B), Sb > := eval(E, S) .
```

While this definitional style is clearly more compact than the previous one, it makes heavy use of matching in conditions [10]. Note that, unlike Maude, many equational/rewriting engines do not provide support for matching in conditions.

The drawback of both definitions above is their lack of explicit support for control information. Indeed, both of them focus on the manipulation of data information in a language, relegating the control flow to the logical infrastructure (equations and/or conditional equations). This is not an issue for the two simple statements defined above, but it may become quite problematic for more control-sensitive statements, such as `halt`, `break` or `continue` statements, not to mention exceptions. To see this, consider adding a `halt` statement to the language:

```
op halt : -> Exp .
```

The semantics of `halt` is the expected one: it stops the evaluation of the program. Note that `halt` can appear anywhere in the program, including functions or loops, which means that the definition of each language construct should be ready to stop immediately if any of its subexpressions evaluates to `halt`. To achieve this, we add a special value for forced termination, say `*`, and then *modify the existing semantics of each language construct* to propagate it accordingly; for example:

```
ceq eval(E1 + E2, S) = < int(I1 + I2), S2 >
   if < int(I1), S1 > := eval(E1, S)
   /\ < int(I2), S2 > := eval(E2, S1) .

ceq eval(E1 + E2, S) = < *, S1 >
   if < *, S1 > := eval(E1, S) .

ceq eval(E1 + E2, S) = < *, S2 >
   if < int(I1), S1 > := eval(E1, S)
   /\ < *, S2 > := eval(E2, S1) .
```

Clearly, this is very inconvenient for at least two reasons: (1) it violates the spirit of modularity in language definitions; adding a new, apparently innocent, language construct requires adding as many additional equations for each language construct as subexpressions it involves; and (2) it does not entirely model the intended meaning of `halt`, namely to abort the execution *immediately*; it still needs to propagate the "halt signal" through all the expressions already started for evaluation, which may be a serious issue if a language with timing constraints is to be defined.

[10] In Maude, conditions in conditional equations are joined together with a conjunction operator /\ and can have extra variables in "matching conditions" of the form $t := u$, which are equational conditions in which a constructor-based pattern t with new variables is matched to the result of evaluating the corresponding instance of u (see [16] for a detailed exposition).

Continuation-Based Definitions

Many programming languages provide statements which are very control intensive, in the sense that they allow one to "jump" through the code/behavior of the program in ways that sometimes may look rather "unstructured". Such control statements include returns from functions, halt, break or continue of loops, and exceptions. Some languages, such as Scheme, even allow their users to "freeze" control contexts and then pass them around just like any other values in the language (`call/cc`), giving programmers a language construct that is more powerful than exceptions or even arbitrary "goto" statements.

The above suggests that in order to naturally define such programming language control-intensive features in rewriting logic, one may need to include as part of the state, besides the data universe of the programming language, also its *control universe*. In other words, one would like to capture "where the execution of the program currently is" as part of some special state attribute to ease the definition of the semantics of certain programming language features. It turns out that, in most cases, a methodological use of *continuations* can enormously ease the definition of the programming language under consideration, at the same time increasing the efficiency of its execution. Note that, unlike higher-order functional approaches, our continuations are first-order structures, resembling quite closely stacks. A continuation-based definitional style will be discussed in depth and illustrated with more examples in Section 3.1. Here we just show how the semantics of addition, conditional and halt statements can be defined with continuations, assuming for now that all the appropriate state infrastructure is already defined (it will be rigorously defined shortly):

```
eq k(exp(E1 + E2) -> K) = k(exp(E1,E2) -> + -> K) .
eq k(val(int(I1),int(I2)) -> + -> K) = k(val(int(I1 + I2)) -> K) .

eq k(exp(if E then E1 else E2) -> K) = k(exp(E) -> if(E1,E2) -> K) .
eq k(val(bool(true))  -> if(E1,E2) -> K) = k(exp(E1) -> K) .
eq k(val(bool(false)) -> if(E1,E2) -> K) = k(exp(E2) -> K) .

eq k(exp(halt) -> K) = k(value(nothing) -> stop) .
```

In the above, the variable `K` ranges over continuations, and the operation `k(...)` wraps a continuation. Continuations can be built by placing continuation items on top of existing continuations using the "stacking" operation:

```
op _->_ : ContinuationItem Continuation -> Continuation .
```

The operations `exp(...)` and `val(...)` wrap lists of expressions and of values, respectively, into continuation items. Appropriate equations evaluate the list of expressions one by one into a list of values, propagating the side effects correspondingly. The constant `stop` is the starting continuation item; there are equations terminating the execution of the program whenever the

continuation has the form in the last equation above, namely a value on top of `stop`.

Section 3.1 below shows how a simple imperative language can be entirely defined in rewriting logic using a continuation-based definitional style.

3.1 A Simple Example

In this section, we illustrate the continuation-based definitional style by means of SIMPLE, a Simple IMPerative LanguagE. SIMPLE is a C-like language, whose programs consist of function declarations. The execution of SIMPLE programs starts by calling the function `main()`. Besides allowing (recursive) functions and other common language features (loops, assignments, conditionals, local and global variables, etc.), SIMPLE is a *multithreaded* programming language, allowing its users to dynamically create, destroy and synchronize threads.

Our language definitions are modular, in the sense that adding or dropping a particular language feature does not affect the definitions of the other features. Each language feature consists of two subparts, its *syntax* and its *semantics*. We define each of the two subparts as separate Maude modules, the latter importing the former. For clarity, we prefer to first define all the syntactic components of the language features, then the necessary state infrastructure, and finally the semantic components.

SIMPLE *Syntax*

Since Maude provides a parser generator for user-defined, context-free [11] mix-fix syntax, we can define the syntax of our programming languages in Maude and use its parser generator to parse programs. We show below how to define the syntax of SIMPLE using mix-fix notation. One should not underestimate the importance and the difficulty of defining the syntax of a programming language. In our experience defining programming languages, getting the syntax of a programming language right may sometimes consume half or even more of our efforts.

We start by defining *names*, or *identifiers*, which will be used as variable or function names. Maude's built-in `QID` module provides us with an unbounded number of quoted identifiers, e.g., `'abc123`, so we can import those and declare `Qid` a subsort of `Name`. Besides the quoted identifiers, we can also define

[11] A context-free grammar can be specified as an order-sorted signature Σ: the sorts exactly correspond to nonterminals; and the mix-fix operator declarations and subsort declarations exactly correspond to grammar productions. Since in Maude each module is either an MEL theory or a rewrite theory, its signature part Σ specifies a user-defined context-free grammar for which Maude automatically generates a parser.

several common names as constants, so we can omit the quotes to enhance the readability of our programs:

```
fmod NAME is
  including QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm
```

SIMPLE is an *expression language*, meaning that everything parses to an expression. As discussed in Section 4.2, complex type checkers can be easily defined on top of the expression syntax if needed. By making use of sorts, it would be straightforward to define different syntactic categories, such as statements, arithmetic and boolean expressions, etc. However, we believe that doing so would decrease the modularity of our definitions; indeed, there are languages in which the assignment can be regarded as a statement, as well as languages in which it can be regarded as an expression having the desired side effect; such languages allow, for example, multiple assignments of the form $x = y = z = 0$, etc. We prefer to keep the syntax of the language simple, the role of discarding bad programs being passed to type checkers.

We first define expressions generically as terms of sort `Exp` extending names and Maude's built-in integers. At this moment we do not need/want to know what other language constructs will be added later on:

```
fmod GENERIC-EXP-SYNTAX is
  including NAME .
  including INT .
  sort Exp .
  subsorts Int Name < Exp .
endfm
```

We are now ready to add language features to the syntax of SIMPLE. We start by adding common arithmetic and boolean expressions:

```
fmod ARITHMETIC-EXP-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  ops _+_ _-_ _*_ : Exp Exp -> Exp [ditto] .
  ops _/_ _%_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BOOLEAN-EXP-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  ops true false : -> Exp .
  ops _==’_ _!=’_ _<’_ _>’_ _<=’_ _>=’_ : Exp Exp -> Exp [prec 37] .
  op _and_ : Exp Exp -> Exp [prec 55] .
  op _or_ : Exp Exp -> Exp [prec 59] .
  op not_ : Exp -> Exp [prec 53] .
endfm
```

Note that we do not distinguish between arithmetic and boolean expressions at this stage. This will be considered when we define the semantics of SIMPLE. The attribute `ditto` associated to some of the arithmetic operators says that they inherit the attributes of the previously defined operators with the same name; these operators were imported toghether with the built-in `INT` module. Built-in modules/features are, of course, not necessary in a language definition. However, it is very convenient to reuse existing efficient libraries for basic language features, such as integer arithmetic, instead of defining them from scratch. Overloading built-in operators is practically useful, but it can somtimes raise syntactic/parsing problems. For example, the built-in binary relational operators on integers evaluate to sort `Bool`, which, for technical and personal taste reasons, we do not want to define as a subsort of `Exp`. Consequently, we cannot overload those operators in our SIMPLE language. That is the reason for which we added a backquote to their names in the module above.

Conditionals are indispensable to almost any programming language:

```
fmod IF-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  op if_then_ : Exp Exp -> Exp .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm
```

Assignments and sequential composition are core features of an imperative language. Unlike in C, we prefer to use the less confusing `:=` operator for assignments (as opposed to just `=`, which many consider to be a poor notation):

```
fmod ASSIGNMENT-SYNTAX is
  including  GENERIC-EXP-SYNTAX .
  op _:=_ : Name Exp -> Exp [prec 41] .
endfm

fmod SEQ-COMP-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  op _;_ : Exp Exp -> Exp [assoc prec 100] .
endfm
```

The attribute[12] `assoc` above states that the operation is associative. This is an essentially semantic property; however, we prefer to give it as part of the syntax because Maude's parser makes use of it to eliminate the need for parentheses.

Lists are used several times in the definition of SIMPLE: lists of names are

[12] In Maude the "modulo axioms" A in a MEL theory $(\Sigma, E \cup A, \mu)$ or a rewrite theory $\mathcal{R}$ can include any combination of *associativity*, *commutativity*, and *identity* axioms. They are declared as equational attributes of their corresponding operator with the `assoc`, `comm`, and `id:` keywords. The Maude interpreter then supports rewriting modulo such axioms with equations and rules.

needed for variable and function declarations, lists of expressions are needed for function calls, lists of values are needed for output as a result of the execution of a program. Since we have a natural subsort structure between the element sorts of these different lists, we can define the corresponding list sorts in a "subtype polymorphism" style. We first define the basic module for lists:

```
fmod LIST is
  sort List .
  op nil : -> List .
  op _,_ : List List -> List [assoc id: nil prec 99] .
endfm
```

From now on, each time we need lists of a particular sort `S`, all we need to do is to define a sort `ListS` extending the sort `List` above, together with an overloaded comma operator. In particular, we can define lists of names as follows:

```
fmod NAME-LIST is
  including NAME .
  including LIST .
  sort NameList .
  subsorts Name List < NameList .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [ditto] .
  eq () = nil .
endfm
```

As syntactic sugar, note that in the above module we defined an additional empty list of names operator, `()`, with the same semantics as `nil`. This is because we prefer to write `f()` instead of `f(nil)` when defining or calling functions without arguments.

Blocks allow one to group several statements into just one statement. Additionally, blocks can define local variables for temporary use:

```
fmod BLOCK-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  including NAME-LIST .
  op {} : -> Exp .
  op {_} : Exp -> Exp .
  op {local_;_} : NameList Exp -> Exp [prec 100 gather (e E)] .
endfm
```

The above general definition of blocks does not only provide the user with a powerful construct allowing on-the-fly variable declarations; but it will also ease later on the definition of functions: a function's body is just an ordinary expression; if one needs local variables then one just defines the body of the function to be a block with local variables.

The syntax of loops is straightforward. We allow both `for` and `while` loops:

```
fmod LOOPS-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  op for(_;_;_)_ : Exp Exp Exp Exp -> Exp .
  op while__ : Exp Exp -> Exp .
endfm
```

The results of a computation need to be reported somehow. We introduce a `print` statement for this purpose. The latest version of Maude has built-in socket objects; one could use those and add output and interaction to our interpreters in a declarative way with rewrite rules. However, the semantics of `print` discussed here is one that collects the output in a list of values, which are all reported at the end of the execution.

```
fmod PRINT-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  op print_ : Exp -> Exp .
endfm
```

Lists of expressions will be needed shortly to define function calls:

```
fmod EXP-LIST is
  including GENERIC-EXP-SYNTAX .
  including NAME-LIST .
  sort ExpList .
  subsort Exp NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm
```

We are now ready to define the syntax of functions. Each function has a name, a list of parameters (given as a list of names), and a body expression. A function call is a name followed by a list of expressions. Functions can be enforced to return abruptly with a typical `return` statement. As explained previously, programs should provide a function called `main`, which is where the execution starts from:

```
fmod FUNCTION-SYNTAX is
  including EXP-LIST .
  sort Function .
  op function___ : Name NameList Exp -> Function [prec 115] .
  op __ : Name ExpList -> Exp [prec 0] .
  op return : Exp -> Exp .
  op main : -> Name .
endfm
```

A program can obviously need more functions, which can even be mutually recursive. We define syntax for *sets* of functions. We use sets because their order does not matter at all: each function can see all the other declared functions in its environment. This language design decision simplifies the syntax of SIMPLE:

```
fmod FUNCTION-SET is
```

```
  including FUNCTION-SYNTAX .
  sort FunctionSet .
  subsort Function < FunctionSet .
  op empty : -> FunctionSet .
  op __ : FunctionSet FunctionSet -> FunctionSet
              [assoc comm id: empty prec 120] .
endfm
```

We want to allow dynamic thread creation in SIMPLE, together with some appropriate synchronization mechanism. The `spawn` statement takes any expression and starts a new thread evaluating that expression. Following common sense in multithreading, the child thread inherits the environment of its parent thread; thus, data-races start becoming possible. To avoid race conditions and to allow synchronization in our language, we introduce a simple lock-based policy, in which threads can acquire and release locks:

```
fmod THREAD-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  ops spawn_ lock acquire_ release_ : Exp -> Exp .
endfm
```

We have defined the syntax of all the desired language features of SIMPLE. All that is needed now to define the syntax of programs is to put all these definitions together. A program consists of a set of global variable declarations and of a set of function declarations:

```
fmod SIMPLE-SYNTAX is
  including ARITHMETIC-EXP-SYNTAX .
  including BOOLEAN-EXP-SYNTAX .
  including IF-SYNTAX .
  including ASSIGNMENT-SYNTAX .
  including SEQ-COMP-SYNTAX .
  including BLOCK-SYNTAX .
  including LOOPS-SYNTAX .
  including PRINT-SYNTAX .
  including FUNCTION-SYNTAX .
  including FUNCTION-SET .
  including THREAD-SYNTAX .
  sort Pgm .
  subsort FunctionSet < Pgm .
  op global_;_ : NameList FunctionSet -> Pgm [prec 122] .
endfm
```

To test the syntax one can parse several programs that one would like to execute and analyze later on, when the semantics will also be defined. In our experience, this is a good time to write tens or even hundreds of benchmark programs. We typically spend significantly more time developing and experimenting with such benchmark programs than with the actual definition of the programming language. This is actually one of the big benefits of our modular approach to defining languages: the definition of the language "per

se" becomes relatively straightforward once one understands what one wants for one's language. We believe that this is precisely what a language designer should look for when choosing a supporting tool. This way one can spend most of the project's efforts on the creative parts rather than on technical low-level details.

The following towers of Hanoi program parses to a term of sort `Pgm`:

```
parse (
  function h(x, y, z, n) {
    if (n >=' 1) then {
      h(x, z, y, n - 1) ;
      print(x) ;
      print(z) ;
      h(y, x, z, n - 1)
    }
  }

  function main() {
    local n ;
    n := 5 ;
    h(1, 2, 3, n)
  }
) .
```

The following concurrent program is a SIMPLE version of a deadlock-prone dining philosophers' program. It parses as a well-formed program. Once we define the semantics of SIMPLE, we will be able not only to execute this multithreaded program but also to analyze it, thus detecting a deadlock automatically:

```
parse (
  global n ;

  function f(x) {
    acquire lock(x) ;
      acquire lock(x + 1) ;
        --- eat
      release lock(x + 1) ;
    release lock(x)
  }

  function main() {
    local i ;
    n := 3 ;
    for(i := 1 ; i <' n ; i := i + 1) spawn(f(i)) ;
    acquire lock(n) ;
      acquire lock(1) ;
        --- eat
      release lock(1) ;
    release lock(n)
  }
```

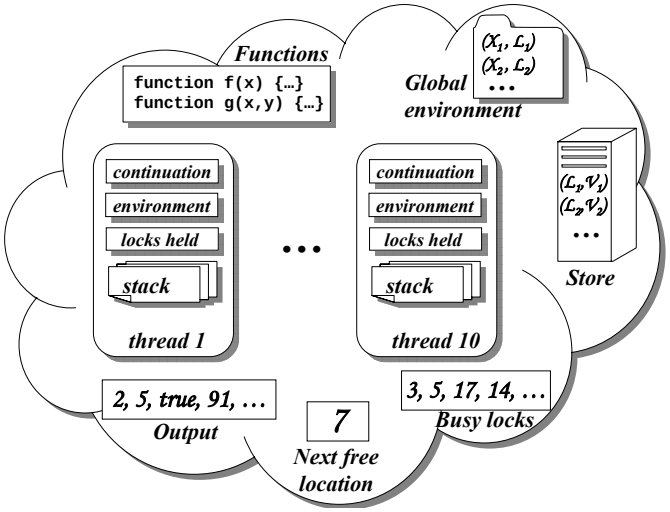

Fig. 2. SIMPLE state infrastructure.

) .

SIMPLE*'s State Infrastructure*

Any practical programming language needs to invariably consider some notion of *state*. The semantics of the various language constructs is defined in terms of how they use or change an existing state. Consequently, before we can proceed to define the semantics of SIMPLE we need to first define its entire state infrastructure.

Figure 2 shows the state infrastructure that we are considering in this paper for the SIMPLE language. All the state ingredients will be explained in detail shortly. Here we just describe them informally. The state can be regarded as a "nested soup", its ingredients being formally called *state attributes*. By "soup" we here mean a multiset with associative and commutative union, and by "nested" we mean that certain attributes can themselves contain other soups inside (for example the threads). Let us next informally describe each of the soup ingredients:

- ***Store.*** The store is a mapping of *locations* into *values*. Each thread will contain its own environment mapping names into locations. Two or more threads can all have access in their environments to the same location in the store, thus potentially causing data-races.
- ***Global environment.*** The global environment maps each global name into a corresponding location. Locations for the global names will be allocated once and for all at the beginning of the execution.
- ***Functions.*** To facilitate (mutually) recursive function definitions, each function sees all the other functions defined in the program. An easy way

to achieve this is to simply keep the set of functions as part of the state.

- ***Next free location.*** This is a natural number giving the next location available to assign a value to in the store. This is needed in order to know where to allocate space for local variables in blocks. Note that in this paper we do not consider garbage collection (otherwise, a more complex schema for the next free location would be needed).
- ***Output.*** The values printed with the `print` statement are collected in an output list. This list will be the result of the evaluation of the program.
- ***Busy locks.*** Thread synchronization in SIMPLE is based on locks. Locks can be acquired or released by threads. However, if a lock is already taken by a thread, then any other thread acquiring the same lock is blocked until the lock is released by the first thread. Consequently, we need to maintain a list of locks that are already busy (taken by some threads); a thread can acquire a lock only if that lock is not in the list of busy locks.
- ***Threads.*** Each thread needs to maintain its own state, because each thread may execute its own code at any given moment and can have its own resources (locations it can access, locks held, etc.). The state of each thread will contain the following ingredients:
  - ***Continuation.*** The tasks/code to be executed by each thread will be encoded as a continuation structure. A continuation will always know, given a value of a local computation step, how to contain the execution of the entire computation that that thread is in charge of.
  - ***Environment.*** A thread may allocate local variables during its execution. The thread can use these variables in addition to the global ones. The local environment of a thread assigns to each variable that thread has access to a unique location in the store.
  - ***Locks held.*** A set of locks held by each thread needs to be maintained. When a thread is terminated, all locks it holds must be released.
  - ***Stack.*** The execution of a thread may naturally involve (recursive) function calls. To ease the semantic definition of the `return` statement, it is convenient to "freeze" and stack the current control context (continuation) whenever a function is called. Then `return` simply pops the previous control context.

We next define the state infrastructure formally. Since locations are countable, we consider them indexed by natural numbers:

```
fmod LOCATION is
  including INT .
  sort Location .
  op loc : Nat -> Location .
endfm
```

Lists of locations are handy in several places, so we define them here. An operation that generates a list of new locations of a given length is also very

useful later:

```
fmod LOCATION-LIST is
  including LOCATION .
  including LIST .
  sort LocationList .
  subsorts Location List < LocationList .
  op _,_ : LocationList LocationList -> LocationList [ditto] .
  op locs : Nat -> LocationList .
  var N : Nat .
  eq locs(N) = if N == 0 then nil else locs(N - 1), loc(N - 1) fi .
endfm
```

An environment is a mapping of names into locations. One elegant way to encode such mappings is as sets of pairs. Below we define environments as sets of pairs [name, location], together with appropriate operations for lookup and for block update:

```
fmod ENVIRONMENT is
  including NAME-LIST .
  including LOCATION-LIST .
  sort Env .
  op empty : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: empty] .
  op _[_] : Env Name -> [Location] .
  op _[_<-_] : Env NameList LocationList -> [Env] .
  vars X X' : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq ([X,L] Env)[X] = L .
  eq ([X',L'] Env)[X <- L]
   = if X == X' then [X',L] Env else [X',L'] (Env[X <- L]) fi .
  eq empty[X <- L] = [X,L] .
  eq Env[X,Xl <- L,Ll] = Env[X <- L][Xl <- Ll] .
  eq Env[nil <- nil] = Env .
endfm
```

We next define values and lists of values:

```
fmod VALUE is
  sort Value .
  op nothing : -> Value .
endfm

fmod VALUE-LIST is
  including VALUE .
  sort ValueList .
  subsort Value < ValueList .
  op nil : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: nil] .
endfm
```

Like environments, a store is also a mapping, but one from locations into

values [13]:

```
fmod STORE is
  including LOCATION-LIST .
  including VALUE-LIST .
  sort Store .
  op empty : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: empty] .
  op _[_] : Store Location -> [Value] .
  op _[_<-_] : Store LocationList ValueList -> [Store] .
  vars L L' : Location .  var Mem : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq ([L,V] Mem)[L] = V .
  eq ([L',V'] Mem)[L <- V]
   = if L == L' then [L',V] Mem else [L',V'] (Mem[L <- V]) fi .
  eq empty[L <- V] = [L,V] .
  eq Mem[L,Ll <- V,Vl] = Mem[L <- V][Ll <- Vl] .
  eq Mem[nil <- nil] = Mem .
endfm
```

A *continuation* is generally understood as a means to encode the remaining part of the computation. We use the operation `_->_` to place a new item on top of an existing continuation. If `K` is some continuation and `V` is some value, then the term `val(V) -> K` is read as "the value `V` is passed to the continuation `K`, which hereby knows how to continue the computation". Several continuation items will be defined modularly as we give the semantics of the various language features. For the time being, let us just define the main continuation constructor:

```
fmod CONTINUATION is
  sorts Continuation ContinuationItem .
  op stop : -> Continuation .
  op _->_ : ContinuationItem Continuation -> Continuation .
endfm
```

The following module defines lists of integers; these are needed for the output:

```
fmod INT-LIST is
  including EXP-LIST .
  sort IntList .
  subsorts Int List < IntList < ExpList .
  op _,_ : IntList IntList -> IntList [ditto] .
endfm
```

Sets of integers are needed to define the "busy" state attribute, holding the set of locks that are already acquired but not yet released by threads. Note

[13] With the parametric capabilities of the latest version of Maude, we could have also used a generic parametric module of mappings and instantiate it to obtain both environments and stores.

that, to avoid syntactic conflicts, we used a set constructor operator which is different from comma:

```
fmod INT-SET is
  including INT .
  sort IntSet .
  subsort Int < IntSet .
  op empty : -> IntSet .
  op _#_ : IntSet IntSet -> IntSet [assoc comm id: empty] .
  op _in_ : Int IntSet -> Bool .
  var I : Int .  var Is : IntSet .
  eq I in I # Is = true .
  eq I in Is = false [owise] .
endfm
```

A tricky aspect of locks is that the same thread may attempt to acquire the same lock multiple times. The usual semantics in such situations, including in Java, is non-blocking, that is, the thread is allowed to acquire the same lock as many times as it requests it. However, in order for the thread to release the lock, it needs to actually release it as many times as it acquired it. In order to define this semantics later on, we need infrastructure to count how many times a lock has been acquired by a thread. Thus, each thread will need to know not only the locks it holds, but also how many times it acquired them. The following piece of infrastructure accounts for that:

```
fmod COUNTER-SET is
  including INT-SET .
  sorts Counter CounterSet .
  subsort Counter < CounterSet .
  op empty : -> CounterSet .
  op [_,_] : Int Int -> Counter .
  op __ : CounterSet CounterSet -> CounterSet [assoc comm id: empty] .
  op _-_ : IntSet CounterSet -> IntSet .
  var I : Int .  var Is : IntSet .  var N : Nat .  var Cs : CounterSet .
  eq (I # Is) - ([I,N] Cs) = Is - Cs .
  eq Is - (empty).CounterSet = Is .
endfm
```

We are now ready to formalize the entire state infrastructure shown informally in Figure 2:

```
fmod SIMPLE-STATE is
  sorts SimpleStateAttribute SimpleState
        SimpleThreadStateAttribute SimpleThreadState .
  subsort SimpleStateAttribute < SimpleState .
  subsort SimpleThreadStateAttribute < SimpleThreadState .
  including ENVIRONMENT .
  including STORE .
  including CONTINUATION .
  including INT-LIST .
  including FUNCTION-SET .
  including COUNTER-SET .
```

```
  op empty : -> SimpleState .
  op __ : SimpleState SimpleState -> SimpleState [assoc comm id: empty] .
  op empty : -> SimpleThreadState .
  op __ : SimpleThreadState SimpleThreadState -> SimpleThreadState
           [assoc comm id: empty] .
  op t : SimpleThreadState -> SimpleStateAttribute .
  op k : Continuation -> SimpleThreadStateAttribute .
  op stack : Continuation -> SimpleThreadStateAttribute .
  op holds : CounterSet -> SimpleThreadStateAttribute .
  op nextLoc : Nat -> SimpleStateAttribute .
  op mem : Store -> SimpleStateAttribute .
  op output_ : IntList -> SimpleStateAttribute .
  op globalEnv : Env -> SimpleStateAttribute .
  op busy : IntSet -> SimpleStateAttribute .
  op functions_ : FunctionSet -> SimpleStateAttribute .
endfm
```

SIMPLE *Semantics*

We are now ready to start defining the semantics of SIMPLE. Several operations are used several places in what follows, so we prefer to define them once and for all at the beginning. The continuation items `exp` and `val` below will be used in the semantics of almost all the SIMPLE language constructs:

```
mod SIMPLE-HELPING-OPERATIONS is
  including NAME-LIST .
  including EXP-LIST .
  including SIMPLE-STATE .

  var X : Name .  vars E E' : Exp .  var El : ExpList .  var K :
  Continuation .  vars V : Value .  var Vl : ValueList .  var Xl :
  NameList .  var Env : Env .  var Mem : Store .  var N : Nat .
  var L : Location .  var TS : SimpleThreadState .

  op exp : ExpList Env -> ContinuationItem .
  op val_ : ValueList -> ContinuationItem .
```

The meaning of `exp(E, Env)` on top of a continuation `K`, that is, the meaning of `exp(E, Env) -> K`, is that `E` is the very next "task" to evaluate, in the environment `Env`. Once the expression `E` evaluates to some value `V`, the continuation item `val(V)` is placed on top of the continuation `K`, which will further process it. Note that in the discussion on the various definitional styles preceeding Section 3.1, we have *not* packed the environment with the expression in `exp(...)`. That followed a slightly different continuation-based definitional methodology, namely one in which the environment of a thread leaves as a separate thread attribute. We have used both styles in our language definitions. The advantage of the one with the environment as a separate thread attribute is that the semantics of many language features tends to be more compact and easier to read. However, one needs additional work to ensure that environments are properly recovered after let bindings or function

invocations (also, at least in the current release of Maude, languages defined using this style are usually less efficient when executed).

It is actually going to be quite useful to extend the meaning above to lists of (sequentially-evaluated) expressions and values, respectively:

```
eq k(exp(nil, Env) -> K) = k(val(nil) -> K) .
eq k(exp((E,E',El), Env) -> K)
 = k(exp(E, Env) -> exp((E',El), Env) -> K) .
eq k(val(V) -> exp(El, Env) -> K)
 = k(exp(El, Env) -> val(V) -> K) .
eq k(val(Vl) -> val(V) -> K) = k(val(V,Vl) -> K) .
```

There are typically several statements in a programming language that write values to particular locations in the store (in our `simple` language, for example, assignments write values at specific locations). Note that, in the definition of a concurrent language, the operation of writing a value at a location needs to be a *rewrite rule*, as opposed to an equation. This is because different threads or processes may "compete" to write the same location at the same time, with different choices potentially making a huge difference in the overall behavior of the program. This is not a problem if one is interested in just getting a correct interpreter for one's language, that is, if one is interested in just one possible execution of the definition, but is of crucial importance when one's goal is to *analyze* concurrent programs [14], as will shortly be shown.

```
op writeTo_ : Location -> ContinuationItem .
rl t(k(val(V) -> writeTo(L) -> K) TS) mem(Mem)
=> t(k(K) TS) mem(Mem[L <- V]) .
```

Like writing values at known locations in the store, binding values to names is also a crucial operation in a language definition. Defining this operation involves several steps, such as creating new locations, binding the new names to them in the current environment, and finally writing the values to the newly created locations. It is interesting to note that, despite the fact that binding involves writing the store, it can be completely accomplished using just equations (no rewrite rules). What makes this possible is the observation that the behavior of a program does/should not depend upon which particular location is allocated to a new name. The equations below destroy the confluence of the rewrite system obtained by orienting all the equations into rewrite rules, but what is important is that they do not affect the possible values to which well-formed programs evaluate (in other words, the resulting rewrite system

[14] This may seem like a break of modularity when one transits from a sequential to a concurrent language, because one would need to translate an equation into a rule. Indeed, there should be no surprise that *analysing* a concurrent language, for example model-checking it, leads to additional concerns and technical support. If one is just interested in obtaining a dynamic semantics of the language then one can simply imagine that all statements are rules and regard the "`eq`" and "`rl`" as annotations for program analysis tools.

is confluent only on the important set of terms containing the well-formed programs):

```
  op bindTo : NameList Env -> ContinuationItem .
  op env : Env -> ContinuationItem .
  eq t(k(val(V,Vl) -> bindTo((X,Xl), Env) -> K) TS)
       mem(Mem) nextLoc(N)
   = t(k(val(Vl) -> bindTo(Xl, Env[X <- loc(N)]) -> K) TS)
       mem(Mem [loc(N),V]) nextLoc(N + 1) .
  eq k(val(nil) -> bindTo(Xl, Env) -> K)
   = k(bindTo(Xl, Env) -> K) .
  eq t(k(bindTo((X,Xl), Env) -> K) TS) nextLoc(N)
   = t(k(bindTo(Xl, Env[X <- loc(N)]) -> K) TS) nextLoc(N + 1) .
  eq k(bindTo(nil, Env) -> K) = k(env(Env) -> K) .

  op exp* : Exp -> ContinuationItem .
  eq env(Env) -> exp*(E) -> K = exp(E, Env) -> K .
endm
```

The above `env` operator allows one to temporarily "freeze" a certain environment in the continuation. The operation `exp*` applied to an expression `E` grabs the environment frozen in the continuation and generates the task of evaluating `E` in that environment.

The remaining modules define the continuation-based semantics of the various SIMPLE language constructs, in the same order in which we introduced their syntax previously. The next module defines the semantics of generic expressions, i.e., integers and names. An integer expression evaluates to its integer value, while a name needs to first grab its location from the environment and then its value from the store. Note that the evaluation of a variable, in other words its "read" action, needs to be a rewrite rule rather than an equation. This is because for SIMPLE programs a read of a (shared) variable may compete with writes of the same variable by other threads, with different orderings leading to potentially different behaviors:

```
mod GENERIC-EXP-SEMANTICS is
  including SIMPLE-HELPING-OPERATIONS .
  op int : Int -> Value .
  var I : Int .  var X : Name .  var K : Continuation .
  var Env : Env .  var Mem : Store .  var TS : SimpleThreadState .
  eq k(exp(I, Env) -> K) = k(val(int(I)) -> K) .
  rl t(k(exp(X, Env) -> K) TS) mem(Mem)
  => t(k(val(Mem[Env[X]]) -> K) TS) mem(Mem) .
endm
```

The continuation-based semantics of arithmetic expressions is straightforward. For example, in the case of the expression `E + E'` on top of the current continuation, one generates the task `(E,E')` on the continuation, followed by the task "add them" (formally a continuation item constant `+`). Once the list `(E,E')` is processed (using other equations or rules), i.e., evaluated to a list

of values, in our case of the form `(int(I), int(I'))`, then all that is left to do is to combine these values into a result value for the original expression, in our case `int(I + I')`, and place it on top of the continuation[15]:

```
mod ARITHMETIC-EXP-SEMANTICS is
  including ARITHMETIC-EXP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  var Env : Env .
  ops + - * / % : -> ContinuationItem .
  eq k(exp(E + E', Env) -> K) = k(exp((E,E'),Env) -> + -> K) .
  eq k(val(int(I),int(I')) -> + -> K) = k(val(int(I + I')) -> K) .
  eq k(exp(E - E', Env) -> K) = k(exp((E,E'), Env) -> - -> K) .
  eq k(val(int(I),int(I')) -> - -> K) = k(val(int(I - I')) -> K) .
  eq k(exp(E * E', Env) -> K) = k(exp((E,E'), Env) -> * -> K) .
  eq k(val(int(I),int(I')) -> * -> K) = k(val(int(I * I')) -> K) .
  eq k(exp(E / E', Env) -> K) = k(exp((E,E'), Env) -> / -> K) .
  eq k(val(int(I),int(I')) -> / -> K) = k(val(int(I quo I')) -> K) .
  eq k(exp(E % E', Env) -> K) = k(exp((E,E'), Env) -> % -> K) .
  eq k(val(int(I),int(I')) -> % -> K) = k(val(int(I rem I')) -> K) .
endm
```

The semantics of the boolean expressions follow precisely the same pattern as that of arithmetic expressions above. Note that the second equation of the semantic definition of each boolean operator can apply only if the evaluation tasks initiated by the first equation yield values of appropriate types:

```
mod BOOLEAN-EXP-SEMANTICS is
  including BOOLEAN-EXP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  op bool : Bool -> Value .
  ops == != < > <= >= and or not : -> ContinuationItem .
  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .  var Env : Env .
  eq k(exp(true, Env) -> K) = k(val(bool(true)) -> K) .
  eq k(exp(false, Env) -> K) = k(val(bool(false)) -> K) .
  eq k(exp(E ==' E', Env) -> K) = k(exp((E,E'), Env) -> == -> K) .
  eq k(val(int(I),int(I')) -> == -> K) = k(val(bool(I == I')) -> K) .
  eq k(exp(E !=' E', Env) -> K) = k(exp((E,E'), Env) -> != -> K) .
  eq k(val(int(I),int(I')) -> != -> K) = k(val(bool(I =/= I')) -> K) .
  eq k(exp(E <' E', Env) -> K) = k(exp((E,E'), Env) -> < -> K) .
  eq k(val(int(I),int(I')) -> < -> K) = k(val(bool(I < I')) -> K) .
  eq k(exp(E >' E', Env) -> K) = k(exp((E,E'), Env) -> > -> K) .
  eq k(val(int(I),int(I')) -> > -> K) = k(val(bool(I > I')) -> K) .
  eq k(exp(E <=' E', Env) -> K) = k(exp((E,E'), Env) -> <= -> K) .
  eq k(val(int(I),int(I')) -> <= -> K) = k(val(bool(I <= I')) -> K) .
  eq k(exp(E >=' E', Env) -> K) = k(exp((E,E'), Env) -> >= -> K) .
  eq k(val(int(I),int(I')) -> >= -> K) = k(val(bool(I >= I')) -> K) .
  eq k(exp(E and E', Env) -> K) = k(exp((E,E'), Env) -> and -> K) .
```

[15] Note the use of the builtin `if-then-else-fi` operator in the last equation. One could easily eliminate it by replacing that equation with two different equations, one for "`val(bool(true))`" and one for "`val(bool(false))`"

```
  eq k(val(bool(B),bool(B')) -> and -> K) = k(val(bool(B and B')) -> K) .
  eq k(exp(E or E', Env) -> K) = k(exp((E,E'), Env) -> or -> K) .
  eq k(val(bool(B),bool(B')) -> or -> K) = k(val(bool(B or B')) -> K) .
  eq k(exp(not E, Env) -> K) = k(exp(E, Env) -> not -> K) .
  eq k(val(bool(B)) -> not -> K) = k(val(bool(not B)) -> K) .
endm
```

We next give the semantics of the conditional statement. The condition is first evaluated, while the two branch expressions are frozen together with the environment in which one of them will need to be evaluated. Once the condition evaluates to a boolean value, the appropriate branch expression is chosen and placed on top of the continuation:

```
mod IF-SEMANTICS is
  including IF-SYNTAX .
  including BLOCK-SYNTAX .
  including BOOLEAN-EXP-SEMANTICS .
  vars BE E E' : Exp .  var K : Continuation .
  var B : Bool .  var Env : Env .
  op if : Exp Exp Env -> ContinuationItem .
  eq if BE then E = if BE then E else {} .
  eq k(exp(if BE then E else E', Env) -> K)
   = k(exp(BE,Env) -> if(E,E',Env) -> K) .
  eq k(val(bool(B)) -> if(E,E',Env) -> K)
   = k(exp(if B then E else E' fi, Env) -> K) .
endm
```

The semantics of the assignment statement is now straightforward, because in the module `GENERIC-EXP-SEMANTICS` we have already defined the semantics of `writeTo`:

```
mod ASSIGNMENT-SEMANTICS is
  including ASSIGNMENT-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var K : Continuation .
  var Env : Env .
  eq k(exp(X := E, Env) -> K)
   = k(exp(E, Env) -> writeTo(Env[X]) -> val(nothing) -> K) .
endm
```

Sequential composition is a common language feature which is used for its side effects. Thus, the semantics of `E ; E'` is that `E` is first evaluated, then its result value is discarded, then `E'` is evaluated; the latter value is the result of the evaluation of the entire sequential composition expression `E ; E'`:

```
mod SEQ-COMP-SEMANTICS is
  including SEQ-COMP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  op discard : -> ContinuationItem .
  vars E E' : Exp .  var K : Continuation .
  var V : Value .  var Env : Env .
  eq k(exp((E ; E'), Env) -> K)
```

```
   = k(exp(E, Env) -> discard -> exp(E', Env) -> K) .
  eq k(val(V) -> discard -> K) = k(K) .
endm
```

Recall that there are three kinds of blocks: an empty block {}, a block without local variable declarations `E`, and a general block with local variable declarations `local Xl ; E`. The first evaluates to the special value `nothing`. The third is obviously more general than the second. We capture this "more general" intuition by actually reducing the second to the third via an equation (the second equation below). The semantics of general blocks with locals is the expected one, that is, the local names are bound to new locations on top of the current environment, and then the body expression of the block is evaluated within the newly obtained environment:

```
mod BLOCK-SEMANTICS is
  including BLOCK-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  var E : Exp .  var K : Continuation .
  var Env : Env .  var Xl : NameList .
  eq k(exp({}, Env) -> K) = k(val(nothing) -> K) .
  eq {E} = {local nil ; E} .
  eq k(exp(({local Xl ; E}), Env) -> K)
   = k(bindTo(Xl,Env) -> exp*(E) -> K) .
endm
```

To simplify the semantic definition of loops, we first note that `for` loops are a special case of `while` loops (captured by the first equation); and second we note that a `while` loop is equivalent to a conditional statement having just one branch which contains the `while` loop (captured by the second equation below). Interestingly, this captures both the fixed-point semantics (if one regards the language specification denotationally, in an initial model semantics style) of loops and its operational semantics (if one attempts to execute the specification):

```
mod LOOPS-SEMANTICS is
  including LOOPS-SYNTAX .
  including IF-SYNTAX .
  including SEQ-COMP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  vars Start Cond Step Body : Exp .  var Env : Env .
  eq for(Start ; Cond ; Step) Body
   = Start ; while Cond (Body ; Step) .
  eq exp(while Cond Body, Env)
   = exp(if Cond then (Body ; while Cond Body), Env) .
endm
```

The semantics of printing is straightforward; the only thing worth mentioning is that the writing of the output "buffer" needs to be a rewrite rule, not an equation. This is because the output is shared by the various threads, so writing an output is similar to writing the store:

```
mod PRINT-SEMANTICS is
  including PRINT-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  op print : -> ContinuationItem .
  var E : Exp . var Env : Env . var K : Continuation .
  var I : Int . var Il : IntList . var TS : SimpleThreadState .
  eq k(exp(print(E), Env) -> K) = k(exp(E, Env) -> print -> K) .
  rl t(k(val(int(I)) -> print -> K) TS) output(Il)
  => t(k(val(nothing) -> K) TS) output(Il,I) .
endm
```

We next give the semantics of function calls. One can regard a function call as an abrupt change of control: the current control context is frozen, then the control is passed to the body of the function; if a `return` statement is encountered, then the frozen control context in which the function call took place is unfrozen and becomes the active one. Since function calls can be nested, the frozen control context needs to be stacked appropriately. This is the reason why we use the thread state attribute called `stack`. The module below should now be self-explanatory:

```
mod FUNCTION-SEMANTICS is
  including FUNCTION-SET .
  including BLOCK-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  op apply : Name -> ContinuationItem .
  op return : -> ContinuationItem .
  op freeze : Continuation -> ContinuationItem .
  var F : Name .  vars Xl LXl : NameList .  var E : Exp .
  var El : ExpList .  var Env : Env .  var TS : SimpleThreadState .
  vars K K' Stack : Continuation .  var V : Value .
  var Vl : ValueList .  var Fs : FunctionSet .

  eq k(exp(F(El), Env) -> K) = k(exp(El, Env) -> apply(F) -> K) .
  eq t(k(val(Vl) -> apply(F) -> K)
     stack(Stack) TS)
     globalEnv(Env) functions(Fs (function F(Xl) {local (LXl) ; E}))
   = t(k(val(Vl) -> bindTo((Xl,LXl), Env) -> exp*(E) -> return -> stop)
     stack(freeze(K) -> Stack) TS)
     globalEnv(Env) functions(Fs (function F(Xl) {local (LXl) ; E})) .

  eq k(exp(return(E), Env) -> K) = k(exp(E, Env) -> return -> K) .
  eq k(val(V) -> return -> K) stack(freeze(K') -> Stack)
   = k(val(V) -> K') stack(Stack) .
endm
```

Let us next define the last and most complex feature of SIMPLE: threads. Creating a new thread is easy: all one needs to do is to add one more term of the form `t(...)` to the top level soup. The newly created term should encapsulate all the corresponding thread attributes. Note that we use a different "stopping" continuation for the newly created threads, called `die`. The meaning of `die` is that threads simply die when they reach it. The reason for not

using the previously defined `stop` continuation is that one sometimes needs to distinguish the first thread from the others (to know when the multi-threaded program terminates, for example):

```
mod THREAD-SEMANTICS is
  including THREAD-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  op lockv : Int -> Value .
  op die : -> Continuation .
  ops lock acquire release : -> ContinuationItem .
  var E : Exp .  var K : Continuation .  var Env : Env .
  var TS : SimpleThreadState .  var V : Value .  var Cs : CounterSet .
  var Is : IntSet .  var N : Nat .  var Nz : NzNat .  var I : Int .

  eq t(k(exp(spawn(E), Env) -> K) TS)
   = t(k(val(nothing) -> K) TS)
     t(k(exp(E, Env) -> die) stack(stop) holds(empty)) .

  eq t(k(val(V) -> die) holds(Cs) TS) busy(Is) = busy(Is - Cs) .
```

Threads without some mechanism for synchronization are close to useless. We chose one of the simplest for SIMPLE, namely one based on locks. Since one would like to evaluate and possibly pass locks around just like any other values in the language, we add a new type of value to the language with the appropriate meaning:

```
  eq k(exp(lock(E), Env) -> K) = k(exp(E, Env) -> lock -> K) .
  eq k(val(int(I)) -> lock -> K) = k(val(lockv(I)) -> K) .
```

Acquiring and releasing locks is quite tricky, because one has to understand very well how these operations interact with concurrency and with multiple acquisitions of the same lock. Indeed, a thread may acquire the same lock more than once; in practice, this situation typically appears when the statement of acquiring a lock is part of a recursive function, in such a way that each recursive function invocation results in acquiring the same lock. Before physically releasing a lock to the runtime environment, one should make sure that the thread requests releasing it as many times as it acquired that lock. This is the semantics of locking in most multithreaded languages, including Java. An important observation here is that, once a thread already holds a given lock, subsequent acquisitions of the same lock are purely local operations that cannot affect the execution of the other threads. Therefore, we can define subsequent lock acquiring using an equation rather than a rule. However, note that the first acquisition of the lock must be defined using a rule, whereas the release can be defined entirely with equations:

```
  eq k(exp(acquire(E), Env) -> K) = k(exp(E, Env) -> acquire -> K) .
  eq k(val(lockv(I)) -> acquire -> K) holds([I, N] Cs)
   = k(val(nothing) -> K) holds([I, N + 1] Cs) .
 crl t(k(val(lockv(I)) -> acquire -> K) holds(Cs) TS) busy(Is)
  => t(k(val(nothing) -> K) holds([I, 0] Cs) TS) busy(I # Is)
```

```
    if not(I in Is) .

  eq k(exp(release(E), Env) -> K) = k(exp(E, Env) -> release -> K) .
  eq k(val(lockv(I)) -> release -> K) holds([I, Nz] Cs)
   = k(val(nothing) -> K) holds([I, Nz - 1] Cs) .
  eq t(k(val(lockv(I)) -> release -> K) holds([I, 0] Cs) TS) busy(I # Is)
   = t(k(val(nothing) -> K) holds(Cs) TS) busy(Is) .
endm
```

We have defined all the features that we want to include in our language. The only thing left to do is to put everything together. We do this by including all the features and defining an `eval` operation on programs, whose result is a list of integers (the output generated with the `print` command):

```
mod SIMPLE-SEMANTICS is
  including SIMPLE-SYNTAX .
  including ARITHMETIC-EXP-SEMANTICS .
  including BOOLEAN-EXP-SEMANTICS .
  including IF-SEMANTICS .
  including ASSIGNMENT-SEMANTICS .
  including SEQ-COMP-SEMANTICS .
  including BLOCK-SEMANTICS .
  including LOOPS-SEMANTICS .
  including PRINT-SEMANTICS .
  including FUNCTION-SEMANTICS .
  including THREAD-SEMANTICS .

  op eval_ : Pgm -> [IntList] .
```

Note that the `eval` operation above actually returns a kind. That is because a program may not always evaluate properly. For example, a program may not be well-typed (a type-checker could remove this worry), may terminate unexpectedly (division by zero), or may not terminate. We define the semantics of `eval` using an auxiliary operation which creates the appropriate initial state. The program terminates when its main thread terminates, that is, when a value (most likely `nothing`) is passed to the starting continuation, `stop`:

```
  op [_] : SimpleState -> [IntList] .

  var Fs : FunctionSet .  var Xl : NameList .  var S : SimpleState .
  var V : Value .  var Il : IntList .  var TS : SimpleThreadState .

  eq eval(Fs) = eval(global nil ; Fs) .
  eq eval(global Xl ; Fs)
   = [t(k(exp(main(), empty) -> stop) stack(stop) holds(empty))
      globalEnv(empty[Xl <- locs(#(Xl))]) nextLoc(#(Xl))
      mem(empty) output(nil) busy(empty) functions(Fs)] .

  eq [t(k(val(V) -> stop) TS) output(Il) S] = Il .
```

In the above we used the length operation `#`. Since we only need it here,

we define it also as part of the same module.

```
  op #_ : ExpList -> Nat .
  var X : Name .
  eq #(X,Xl) = 1 + #(Xl) .
  eq #(nil) = 0 .
endm
```

The semantics of SIMPLE is now complete. The first benefit one gets from this definition is that one gets an *interpreter for free*. Indeed, all one needs to do is to start a Maude rewrite session using the command "`rew eval(program)`", where `program` can be any program that parses; for example, the towers of Hanoi or the dining philosophers programs parsed previously. In Section 4.1 we show how one can use the exact same definition of SIMPLE to formally *analyze* programs.

3.2 Other Language Case Studies

The SIMPLE language discussed in Section 3.1 illustrates a particular language specification style; but this is just one example within a much broader language specification methodology. A key point worth making is that this methodology *scales up* quite well to real languages with many features, both in terms of still allowing very readable and understandable specifications, and also in being capable of providing high performance interpreters and competitive program analysis tools. For example, large fragments of Java and the JVM have been specified in Maude this way, with the Maude rewriting logic semantics being used as the basis of Java and JVM program analysis tools that for some examples outperform well-known Java analysis tools [26,24]. A similar Maude specification of the semantics of Scheme at UIUC yields an interpreter with .75 the speed of the standard Scheme interpreter on average for the benchmarks we have tested. In fact, the semantics of large fragments of conventional languages are routinely developed by UIUC graduate students as course projects in a few weeks, including, besides Java, the JVM, and Scheme, languages like (alphabetically), Beta, Haskell, Lisp, LLVM, Pict, Python, Ruby, and Smalltalk. A semantics of a Caml-like language with threads was discussed in detail in [41], and a modular rewriting logic semantics of a subset of CML has been given by Chalub and Braga in [12]. Following a continuation-based semantics similar to the one in this paper, d'Amorim and Roşu have given a definition of the Scheme language in [21]. Other language case studies, all specified in Maude, include BC [7], CCS [63,64,7], CIAO [58], Creol [34], ELOTOS [61], MSR [10,56], PLAN [57,58], and the pi-calculus [59].

4 Program Analysis Techniques and Tools

Specifying formally the rewriting logic semantics of a programming language in Maude yields a prototype interpreter for free. Thanks to generic analysis tools for rewriting logic specifications currently provided as part of the Maude system, we additionally get the following analysis tools also *for free*:

(i) a *semi-decision procedure* to find failures of safety properties in a (possibly infinite-state) concurrent program using Maude's `search` command;

(ii) an LTL *model checker* for finite-state programs or program abstractions;

(iii) a *theorem prover* (Maude's ITP [17,18]) that can be used to prove programs correct semi-automatically.

We discuss the first two items, and also a generic partial order reduction tool under development [25], in Section 4.1, where we give some examples illustrating this kind of automated analysis for programs in SIMPLE. Analyses based on abstract semantics are discussed in Section 4.2. The relationship to logics of programs and to semantics-based theorem proving, including uses of the Maude ITP, is discussed in Section 4.3. Modularity and the MSOS Tool are discussed in Section 4.4.

4.1 Search and Model Checking Analysis

In this section we illustrate the search and model checking analysis capabilities that one obtains for free from a rewrite logic semantic definition of a programming language. Let us consider again the definition of SIMPLE, together with the dining philosophers program in Section 3.1. If one executes that program using the command `rew eval(program)` then most likely one would see a normal execution, that is, one which terminates and outputs nothing. That is because there is a very small likelihood that the program will deadlock. Nevertheless, the potential for deadlock is there, meaning that some other executions of the same program may deadlock, with all the usual, undesired consequences.

To analyze all the possible rewriting computations from an initial state in a given rewriting logic specification, Maude provides a `search` command. This command takes an initial state to analyze, a pattern to be reached, and optionally a semantic condition to be satisfied by the reached pattern, and searches through all the state space generated in a breadth-first manner, by considering all the different rewrite rules that can be applied to each reachable state. Once one defines a rewriting logic specification of a language in Maude, one can simply use the built-in search capabilities of Maude to exhaustively search for executions of interest through the state space of a given program. The following search command generates all the states in which the dining philosophers program can deadlock:

```
search eval(
  global n ;
  function f(x) {
    acquire lock(x) ;
      acquire lock(x + 1) ;
        --- eat
      release lock(x + 1) ;
    release lock(x)
  }
  function main() {
    local i ;
    n := 3 ;
    for(i := 1 ; i <' n ; i := i + 1) spawn(f(i)) ;
    acquire lock(n) ;
      acquire lock(1) ;
        --- eat
      release lock(1) ;
    release lock(n)
  }
) =>! Il:[IntList] .
```

The suffix `... =>! Il:[IntList]` tells the `search` command to search for all the normal forms of kind `[IntList]`, that is, all the normal forms of that program. As expected, the above returns *two* normal forms: one in which the program terminates and one in which each thread acquired one lock and is waiting, in a deadlock, for the other one to be released. Note that in both this states the state attribute `output` contains an empty list of values.

When a deadlock is detected in a concurrent program, one is normally expected to fix it. A common fix for the dining philosophers deadlock is to force the philosophers to follow a certain discipline in acquiring the forks: philosophers on odd positions acquire the left fork first and then the right one, while philosophers on even positions take the right fork first followed by the second. The following deadlock-free version of dining philosophers can be verified as follows:

```
search eval(
  global n ;
  function f(x) {
    if x % 2 ==' 1
    then {
      acquire lock(x) ;
        acquire lock(x + 1) ;
          --- eat
        release lock(x + 1) ;
      release lock(x)
    }
    else {
      acquire lock(x + 1) ;
        acquire lock(x) ;
          --- eat
        release lock(x) ;
```

```
      release lock(x + 1)
    }
  }

  function main() {
    local i ;
    n := 3 ;
    for(i := 1 ; i <' n ; i := i + 1) spawn(f(i)) ;
    if n % 2 ==' 1
    then {
      acquire lock(n) ;
        acquire lock(1) ;
          --- eat
        release lock(1) ;
      release lock(n)
    }
    else {
      acquire lock(1) ;
        acquire lock(n) ;
          --- eat
        release lock(n) ;
      release lock(1)
    }
  }
) =>! Il:[IntList] .
```

As expected, the above search returns only one solution, the one stating a normal termination of the concurrent program. That means that the fork-grabbing strategy above indeed fixes the deadlock bug.

The above-mentioned deadlock is not the only flaw with the original dining philosophers program. Consider the slightly modified version of dining philosophers where each philosopher continues to alternatively think and eat forever. Another property worth checking for this program is to see whether a certain philosopher (say philosopher 3) starves or not. To check this, it suffices to define a parametric predicate `eaten(i)` which holds in the state where philosopher `i` is eating. Then, using Maude's built-in LTL model checker, one can simply check whether the LTL formula `[]<> eaten(i)` holds or not as follows:

```
red modelCheck(eval(
  global n ;
  function f(x) {
    while(true) {
      if x % 2 ==' 1
      then {
        acquire lock(x) ;
          acquire lock(x + 1) ;
            --- eat
          release lock(x + 1) ;
        release lock(x)
      }
```

```
      else {
        acquire lock(x + 1) ;
          acquire lock(x) ;
            --- eat
          release lock(x) ;
        release lock(x + 1)
      }
      --- think
    }
  }

  function main() {
    local i ;
    n := 3 ;
    for(i := 1 ; i <' n ; i := i + 1) spawn(f(i)) ;
    while (true) {
      if n % 2 ==' 1
      then {
        acquire lock(n) ;
          acquire lock(1) ;
            --- eat
          release lock(1) ;
        release lock(n)
      }
      else {
        acquire lock(1) ;
          acquire lock(n) ;
            --- eat
          release lock(n) ;
        release lock(1)
      }
    }
    --- think
  }
), []<> eaten(3)) .
```

which as one expects returns a counterexample in which philosophers 1 and 2 keep eating alternatively, and philosopher 3 never gets a chance to eat.

It is well-known that concurrency leads to massive increases in the state space of a program, because there are very many equivalent interleavings of the same computation that have to be checked by a standard model checker. One way to avoid this state explosion is to use *partial order reduction* (POR) techniques (see [15] and references there), in which many of these interleaving computations are never explored. POR is *complete*, in the sense that an LTL formula not involving the "next" operator $\bigcirc$ can be shown to hold using POR model checking iff it can be shown to hold using standard model checking [15]. The traditional way to provide a POR model checking capability for a given programming language $\mathcal{L}$ is to *modify the model checking algorithm* of a model checker for $\mathcal{L}$. This is a substantial task, which furthermore has to be performed for each different language. Since we are interested in amortizing

the cost of *all* program analysis tools across many languages by making them generic, A. Farzan and the first author are currently exploring a POR model checking technique [25] that is generic in the language $\mathcal{L}$, under very general assumptions about $\mathcal{L}$, such as having processes or threads endowed with unique identities. An important advantage of this language-generic POR technique is that it does not require any changes to an underlying model checker. In particular, it can be used together with the Maude LTL model checker to model check with POR programs in any programming language $\mathcal{L}$ satisfying a few general assumptions. The key idea is to perform a *theory transformation* of the rewrite theory $\mathcal{R}_{\mathcal{L}}$ specifying the semantics of $\mathcal{L}$ to obtain a POR-enabled, semantically equivalent rewrite theory $\mathcal{R}_{\mathcal{L}}^{por}$. We can then use a standard LTL model checker to model check programs in $\mathcal{L}$ with POR reduction by model checking them in a standard way using $\mathcal{R}_{\mathcal{L}}^{por}$. Initial experiments with semantic definitions for the JVM and a Promela-like language suggest that the ratios of state-space reduction obtained with this generic POR technique are comparable to those reported using language-specific model checkers with a built-in POR capability [25].

4.2 Analyses Based on Abstract Semantics

The three types of analyses discussed so far, namely interpretation/simulation, search and model checking, make use of the semantic rewriting logic definition of a programming language *as is*. Therefore, a language designer obtains all these analysis capabilities essentially *for free*. There are, however, certain kinds of analysis that require a slightly different, typically more abstract semantics to be defined. One should *not* regard the need for a different semantics as a break of modularity, but rather as defining a totally different system, or "language", namely one that "interprets" the syntax differently. Interestingly, one can do this relatively easily, by just rewriting the existing language semantics appropriately.

The already existing semantic definition of the language acts as a check-list telling the analisys tool developer *what* needs to be defined and only partly *how* to define it. The tool developer is responsible for filling in all the details. In the case of simple analysers, such as a type checker or an abstract interpreter in which the abstract domain and its properties can be inferred from the concrete domain in some straightforward manner, one can immagine automatic generators of analysis tools, by providing some general rules stating how the concrete semantics needs to be changed. While this is clearly an interesting research subject, we do not pursue it here. We assume that the tool developer is responsible for the entire definition of the analser, admitting that defining some parts of it can be uninteresting or even boring. In this section we briefly discuss two kinds of static analysis tools that we have experimented with, namely type checkers and domain-specific certifiers.

Let us first elaborate on some intuitions underlying the definition of a type checker. To keep the discussion focused, let us assume a type checker for SIMPLE. Since a type checker is not concerned with the concrete values handled by a program, but instead with their types, we replace values in the definition of SIMPLE by types. The continuation item `val(...)` becomes `type(...)` and several constant types need to be added, such as `int`, `bool`, etc. Recall the continuation-based definition of comparison:

```
eq k(exp(E >' E', Env) -> K) = k(exp((E,E'), Env) -> > -> K) .
eq k(val(int(I),int(I')) -> > -> K) = k(val(bool(I > I')) -> K) .
```

Viewed through the prism of types, the above says that `E >' E'` has the type `bool` if `E` and `E'` have the type `int`. It is then straightforward to modify the above as follows:

```
eq k(exp(E >' E', Env) -> K) = k(exp((E,E'), Env) -> > -> K) .
eq k(type(int,int) -> > -> K) = k(type(bool) -> K) .
```

Of course, environments in the concrete semantics become type environments in the abstract semantics, assigning types to names. One can systematically modify the semantics of each language construct as above, thereby easily obtaining a type checker. The abstract semantics of some language constructs can be almost automatically derived from the concrete semantics, like in the case above, but there may also be some language constructs whose abstract semantics needs some thinking. This is because one may need to consider certain trade-offs in order to obtain an effective tool. For example, the concrete semantics of a conditional did not care about the types of the two branches:

```
eq k(exp(if BE then E else E', Env) -> K)
 = k(exp(BE,Env) -> if(E,E',Env) -> K) .
eq k(val(bool(B)) -> if(E,E',Env) -> K)
 = k(exp(if B then E else E' fi, Env) -> K) .
```

Also, the way it was defined, the conditional can be used either as a statement or as an expression, without any explicit definitional support. One can even use it in contexts in which the type of some branch is `int`, while the type of the other is `bool`. However, in order to be effective, a type checker imposes restrictions on how the language constructs can be used. Whether a certain set of restrictions is reasonable or not is a language design decision that we are not concerned with here. Instead, our purpose is to show that one can very easily experiment with such decisions in our framework. One quite reasonable requirement in typing conditionals in a language like SIMPLE is that the condition types to `bool`, while the two branches type to the same type. Thus conditionals are still allowed to be "polymorphic", but their use needs to respect the reasonable restriction that the two branches have the same type. This restriction allows one to assign a unique type to the entire conditional statically, namely the type of its two branches. The type semantics of the conditional would then be changed to:

```
eq k(exp(if BE then E else E', Env) -> K)
 = k(exp((BE,E,E'),Env) -> K) .
eq k(type(bool,T,T) -> K) = k(type(T) -> K) .
```

We have defined several type checkers following this semantic abstraction methodology as part of our programming language courses [50]. Students also developed such type checkers as part of their homework assignments, including ones based on type inference. In the case of type reconstruction, the result of "evaluating" an expression is a set of equational type constraints. All these type constraints are solved either at the end of the evaluation process or on the fly.

Another category of analysis tools that we have investigated, also derived from the semantics of a given programming language, is that of domain-specific certifiers. Like in type checking, expressions evaluate to some abstract values. However, unlike in type checking, these abstract values have no relationship whatsoever with the concrete values. The abstract values make sense only in the context of a specific domain of interest, which also needs to be formally axiomatized or defined. Consider, for example, the domain of units of measurement, which can be formalized as an abelian group generated by the basic units (meter, second, foot, etc.) – suppose that multiplication of units is written as concatenation. A program certifier for this domain would check that, in a program written in an extended syntax allowing annotations specifying the units of variables, all the operations performed by the given program are consistent with the intuitions of the domain of units of measurement. For example, only expressions which have the same unit can be added or compared, while expressions of any units can be multiplied. The semantic definitions of addition and multiplication in a domain-specific certifier for SIMPLE would look something like:

```
eq k(exp(E + E', Env) -> K) = k(exp((E,E'), Env) -> + -> K) .
eq k(unit(U,U) -> + -> K) = k(unit(U) -> K) .

eq k(exp(E * E', Env) -> K) = k(exp((E,E'), Env) -> * -> K) .
eq k(unit(U,U') -> * -> K) = k(unit(U U') -> K) .
```

Formal definitions of domain specific certifiers built on semantic programming language definitions have been investigated in depth in several places. In [14,13] we discuss such certifiers for the domains of units of measurement and a large fragment of C, in [35] we present a domain-specific certifier for the domain of coordinate frames, and in [51] one for the domain of optimal state estimation.

Each analysis tool has its particularities and may raise complex issues, from difficulty in defining it to intractability. The main point we want to stress in this section is that the original rewriting semantics of the programming language gives us a very useful skeleton on which to develop potentially any desired program analysis tool.

4.3 Logics of Programs and Semantics-Based Theorem Proving

Given a programming language $\mathcal{L}$, we are often interested in using a *logic of programs* for $\mathcal{L}$ to reason about programs in $\mathcal{L}$. For example, we may want to use a Hoare logic with inference rules corresponding, say, to sequential Java features to reason about sequential Java programs. Two important tasks appear in this regard:

(i) the correctness of the chosen logic of programs for the given language $\mathcal{L}$ has to be justified in term of a mathematical definition of $\mathcal{L}$'s semantics; and

(ii) *mechanizing* a logic of programs for $\mathcal{L}$ typically requires not only mechanizing the logic's inference rules, but also the discharging of *verification conditions* (VCs) generated by the inference process; and for discharging such VCs one often needs to use properties of $\mathcal{L}$'s underlying semantics.

In our example of a Hoare logic for Java programs, the first task corresponds to showing that the chosen Hoare rules are correct with respect to the Java semantics; and the second to having a way to reason about the truth of VCs, once the Hoare rules have done their job of decomposing the task of proving a Hoare correctness assertion into proving the formulas generated as VCs.

Having a mathematically precise semantics of a programming language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$ can be very useful for tasks (i) and (ii). In some sense, task (i) is the most crucial and fundamental. However, since mistakes in large specifications are a fact of life, task (i) should really be understood as a *mutual debugging process*: the chosen logic of programs can be debugged using the semantics $\mathcal{R}_{\mathcal{L}}$. And $\mathcal{R}_{\mathcal{L}}$ itself, which being a large specification may contain some bugs, can be debugged not only by executing $\mathcal{R}_{\mathcal{L}}$ as an interpreter, but also by using both the chosen logic of programs and $\mathcal{L}$'s informal semantics to ascertain which specification is wrong (that of the logic of programs, or $\mathcal{R}_{\mathcal{L}}$, or both) when discrepancies arise.

An important case study for task (i) , showing the usefulness of the specification $\mathcal{R}_{\mathcal{L}}$ when $\mathcal{L}$ is Java source code, has been recently carried out by W. Ahrendt, A. Roth, and R. Sasse [52,1]. The goal was to validate automatically the correctness of a substantial set (about 50 inference rules) of the JavaCard Dynamic logic [3], a logic for JavaCard programs with diamond and box modalities in which a sentence $< p > \phi$ with p a JavaCard program and ϕ a formula is interpreted as: "p terminates in a state in which ϕ holds". The total correctness interpretation of a Hoare triple $\{\psi\}p\{\phi\}$ is expressed in this logic as the formula $\psi \rightarrow < p > \phi$. The JavaCard Dynamic logic has inference rules, implemented by so-called *taclets*, reducing the proof of a dynamic logic formula to that of simpler such formulas. Of particular interest in this case study were the *code transformation taclets*, which implement inference rules

of the form

$$\frac{< p > \phi}{< p' > \phi}$$

that is, code transformation taclets transform the program part into a simpler equivalent form, leaving the formula ϕ unchanged. Of course, such taclets are *rewrite rules*, in which the subexpressions p, p' and ϕ typically are not concrete JavaCard programs and a concrete formula; they are instead *patterns*, symbolic expressions called program schemes in which part of the scheme is concrete code and the rest – corresponding to additional schematic code – consists of meta-variables which symbolically represent all their concrete code instantiations. Similarly, ϕ may just be a meta-variable of type formula.

The goal of this research was to automatically validate a large set of code transformation taclets by symbolically evaluating the program schemes p and p' using $\mathcal{R}_{Java}$ to check that they were semantically equivalent. This was a non-trivial task, because $\mathcal{R}_{Java}$ can be executed as an interpreter when p and p' are *concrete* Java programs. In the case of program schemes, although symbolic execution is still possible, the axioms in $\mathcal{R}_{Java}$ are insufficient to reason about semantic equivalence. The elegant solution adopted in [52,1] is to lift $\mathcal{R}_{Java}$ to the symbolic level by specifying a more expressive semantics $\mathcal{R}^{lift}_{Java}$. Two key issues making such a lifting necessary are: (1) the need to *localize* semantic equivalence to the program variables shared by p and p', excluding fresh new variables introduced in one of them; and (2) the need to describe symbolically *unknown side effects* caused when symbolically executing the parts of a program scheme described by meta-variables. Using the lifted semantics $\mathcal{R}^{lift}_{Java}$, over 50 code transformation taclets have been automatically validated using Maude. Furthermore, all axiomatic propositional logic taclets have also been automatically validated in Maude [52,1]. One particularly valuable outcome of this substantial case study has been the kind of mutual debugging mentioned above: on the one hand, a bug was found in one of the taclets; on the other, several bugs were also uncovered in the prototype Java semantics reported in [24], which was used as a basis to develop the $\mathcal{R}^{lift}_{Java}$ specification in Maude.

Another substantial case study, this time involving both tasks (i) and (ii), has centered on the Pascal-like language used in [30]. A Hoare logic for a substantial fragment of this Pascal-like language has been given in [39], where the correctness of the Hoare rules is mathematically justified on the basis of the language's formal semantics $\mathcal{R}_{\mathcal{L}}$, thus addressing task (i). Task (ii) has been addressed by M. Clavel and J. Santa-Cruz in [19] using the Maude inductive theorem prover (ITP) as the underlying proof engine. Their ASIP+ITP tool integrates the theory $\mathcal{R}_{\mathcal{L}}$ with the Maude ITP and directly supports some Hoare rules. A user can state goals as Hoare triples; the Hoare rules are then applied by the ASIP tool to generate VCs, which can be discharged using the ITP [19]. The ASIP tool and its documentation can be downloaded from `http://maude.sip.ucm.es/itp/asip/`.

A third relevant case study involves the Hennessy-Milner logic of programs for CCS [64,61]. Since the justification of the Hennessy-Milner logic is well-established, this work focuses on task (ii). A. Verdejo and N. Martí-Oliet first give a rewriting logic semantics for CCS as a rewrite theory $\mathcal{R}_{CCS}$ in Maude. Then, an inference system for the Hennessy-Milner modal logic of CCS is also defined in Maude as another module at the meta-level that imports the module specifying $\mathcal{R}_{CCS}$ at the object level. In this way, the CCS interpreter obtained by the Maude specification of $\mathcal{R}_{CCS}$ is seamlessly extended into a program reasoning tool for CCS, in which the satisfaction of a Hennessy Milner logic formula ϕ by a finitary CCS process P can be automatically verified by the Maude-based tool [64,61]. This work again demonstrates the usefulness of integrating the inference rules of a language's logic of programs with an executable rewriting logic semantics of that language to generate a mechanization of the given logic of programs in a relatively effortless way. As in the case of the ASIP+ITP tool [19], this integration is achieved in its entirety within the rewriting logic framework using reflective techniques. The executable CCS semantics in Maude, and the Hennessy-Milner logic tool are both available in `http://www.ucm.es/sip/alberto/esf/`.

4.4 Modularity and the MSOS Maude Tool

Modularity of semantic definitions, that is, the property that a feature's semantics does not have to be redefined when a language is extended, is notoriously hard to achieve. Lack of modularity has plagued both denotational semantics and SOS. Without a suitable specification *methodology* it could also plague rewriting logic semantics.

One important transfer of results between the SOS and rewriting logic approaches has been precisely on modularity issues. On the one hand, Mosses' *modular structural operational semantics* (MSOS) methodology [47] has inspired Braga and the first author to develop a similar modular methodology for rewriting logic semantics [40,7]. On the other hand, this has had the pleasant side-effect of providing a Maude-based execution environment for MSOS specifications, namely the Maude MSOS Tool developed at the Universidade Federal Fluminense in Brazil by Chalub and Braga [11], which is available on the web at `http://mmt.ic.uff.br`.

At the theoretical level, the point is that, as shown in [40], we can associate to an MSOS specification $\mathcal{S}$ a semantically equivalent, and also modular, rewrite theory $\tau(\mathcal{S})$. This translation τ is then the basis of the Maude MSOS tool. In this way, MSOS specifications, besides being executable as language interpreters, can be used to formally analyze programs in the language so specified using the underlying Maude tools.

Modularity at the rewriting logic level opens up new possibilities not available at the SOS level. For example, as pointed out in [40], the modular

methodology is simultaneously available *for both equations and rewrite rules*. As already pointed out, this useful "abstraction knob" that allows us to distinguish between deterministic and nondeterministic features by respectively specifying their semantics with equations or with rules is important not only conceptually, but also to make formal analyses such as search and model checking much more scalable by drastically reducing the size of the associated search spaces.

5 Conclusions and Future Directions

We have explained how rewriting logic can be used as a logical framework to unify equational semantics and SOS; and how, using a language such as Maude and its generic tools, efficient interpreters and program analysis tools can be generated from language definitions. This paper is just a snapshot of what we believe is a promising collective research project. Much work remains ahead. We list below some future research directions that we find particularly attractive:

- *Modularity.* As mentioned in Section 4.4, a fully modular definitional style has already been developed in [40]. An interesting open question is: what other definitional styles can likewise be endowed with a fully modular methodology? At the experimental level this should lead to a well-crafted library of modular semantic definitions in the spirit of MSOS, so that new language definitions can easily be developed by composing the semantic definitions of their basic features, changing their generic abstract syntax to the concrete syntax of the language in question.
- *Semantic Equivalence and Compiler Generation.* It would be highly desirable to develop general methods to show that two semantic definitions of a programming language are equivalent. Meta-results of this kind could be the basis of automated semantics-preserving translations between language definitions given in different definitional styles. They could also be the basis of generic formal compiler techniques; and of compiler generators that, taking a formal language definition as input, and are provably correct, in the sense of preserving the language's semantics.
- *Generic Tools.* Although some quite useful generic tools already exist, it is clear that much more can be done. For example, it would be quite useful to have a generic *abstraction tool*, so that an infinite-state program in any language satisfying minimal requirements can be model checked by model checking a finite-state abstraction. Similarly, a *language-generic theorem proving tool* allowing the kind of reasoning supported at present by language-specific tools such as ASIP+ITP [19] for a large class of languages would likewise be highly desirable.

Acknowledgement

We cordially thank Wolfgang Ahrendt, Christiano Braga, Feng Chen, Manuel Clavel, Marcelo D'Amorim, Santiago Escobar, Azadeh Farzan, Mark Hills, Narciso Martí-Oliet, Peter Mosses, Andreas Roth, Ralph Sasse, Mark-Oliver Stehr, Traian Florin Şerbănuţă, Carolyn Talcott, Alberto Verdejo, and the UIUC students attending the programming language design and semantics courses [50], who defined many real programming languages, for their various kinds of help with this paper, including examples, careful comments, and their own intellectual contributions to the rewriting logic semantics program. This work has been partially supported by the following grants: ONR N00014-02-1-0715, NSF/NASA CCF-0234524, NSF CAREER CCF-0448501, and NSF CNS-0509321.

References

[1] W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. Manuscript, June 2005.

[2] D. Basin and G. Denker. Maude versus Haskell: an experimental comparison in security protocol analysis. In K. Futatsugi, editor, *Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications*, volume 36. ENTCS, Elsevier, 2000.

[3] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

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

[5] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[6] C. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Departamento de Informática, Pontificia Universidade Católica de Rio de Janeiro, Brasil, 2001.

[7] C. Braga and J. Meseguer. Modular rewriting semantics in practice. In *Proc.* WRLA'04, volume 117. ENTCS, Springer, 2004.

[8] M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Trans. on Prog. Lang. and Systems*, 9(1):54–99, Jan. 1987.

[9] R. Bruni and J. Meseguer. Generalized rewrite theories. In J. Baeten, J. Lenstra, J. Parrow, and G. Woeginger, editors, *Proceedings of ICALP 2003, 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Springer LNCS*, pages 252–266, 2003.

[10] I. Cervesato and M.-O. Stehr. Representing the MSR cryptoprotocol specification language in an extension of rewriting logic with dependent types. In P. Degano, editor, *Proc. Fifth International Workshop on Rewriting Logic and its Applications (WRLA'2004)*, volume 117. Elsevier ENTCS, 2004. Barcelona, Spain, March 27 - 28, 2004.

[11] F. Chalub. An Implementation of Modular SOS in Maude. Master's thesis, Universidade Federal Fluminense, May 2005. `http://www.ic.uff.br/~frosario/dissertation.pdf`.

[12] F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, July 2004. `http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics`.

[13] F. Chen and G. Roşu. Certifying measurement unit safety policy. In *Proceedings, International Conference on Automated Software Engineering (ASE'03)*. IEEE, 2003.

[14] F. Chen, G. Roşu, and R. P. Venkatesan. Rule-based analysis of dimensional safety. In *Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Springer LNCS*, pages 197–207, 2003.

[15] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2001.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[17] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000. `http://maude.cs.uiuc.edu`.

[18] M. Clavel and M. Palomino. The ITP tool's manual. Universidad Complutense, Madrid, April 2005, `http://maude.sip.ucm.es/itp/`.

[19] M. Clavel and J. Santa-Cruz. ASIP + ITP: A verification tool based on algebraic semantics. In *Proc. PROLE'05*, 2005. To appear, `http://maude.sip.ucm.es/~clavel/pubs/`.

[20] D. Clément, J. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. In K. Fuchi and M. Nivat, editors, *Proceedings, France-Japan AI and CS Symposium*, pages 49–89. ICOT, 1986. Also, Information Processing Society of Japan, Technical Memorandum PL-86-6.

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

[22] R. DeNicola and U. Montanari. Selected papers of the 2nd workshop on concurrency and compositionality, San Miniato, Italy, March 1990. *Theoretical Computer Science*, 96(1), 1992.

[23] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

[24] A. Farzan, F. Cheng, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In *Proc. CAV'04*, volume 3114 of *Springer LNCS*, 2004.

[25] A. Farzan and J. Meseguer. Partial order reduction for rewriting semantics of programming languages. Technical Report UIUCDCS-R-2005-2598, CS Dept., University of Illinois at Urbana-Champaign, June 2005.

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

[27] M. Felleisen and D. P. Freidman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III, Proceedings of IFIP TC2 Working Conference*, pages 193–217. North Holland, 1987.

[28] F. Gadducci and U. Montanari. The tile model. In G. Plotkin, C. Stirling and M. Tofte, eds., *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, 133–166, 2000.

[29] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.

[30] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[31] J. A. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Diaz and I. Ramos, editors, *Formalizing Programming Concepts*, pages 292–309. Springer-Verlag, 1981. LNCS, Volume 107.

[32] P. H. Hartel. LETOS – a lightweight execution tool for operational semantics. *Software: Practice and Experience*, 29:1379–1416, 1999.

[33] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Willey & Sons, 1990.

[34] E. B. Johnsen, O. Owe, and E. W. Axelsen. A runtime environment for concurrent objects with asynchronous method calls. In N. Martí-Oliet, editor, *Proc. 5th. Intl. Workshop on Rewriting Logic and its Applications*, volume 117. ENTCS, Elsevier, 2004.

[35] M. Lowry, T. Pressburger, and G. Roşu. Certifying domain-specific policies. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 81–90. IEEE, 2001. Coronado Island, California.

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

[37] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.

[38] J. Meseguer. Software specification and verification in rewriting logic. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logic of Engineering Software, NATO Advanced Study Institute, Marktoberdorf, Germany, July 30 – August 11, 2002*, pages 133–193. IOS Press, 2003.

[39] J. Meseguer. Lecture notes on program verification. CS 476, University of Illinois, `http://www-courses.cs.uiuc.edu/~cs476/`, Spring 2005.

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

[41] J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In *Proc. Intl. Joint Conf. on Automated Reasoning IJCAR'04, Cork, Ireland, July 2004*, pages 1–44. Springer LNAI 3097, 2004.

[42] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[43] J. Moore, R. Krug, H. Liu, and G. Porter. Formal models of Java at the JVM level – a survey from the ACL2 perspective. In *Proc. Workshop on Formal Techniques for Java Programs, in association with ECOOP 2001*, 2002.

[44] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B, Chapter 11*. North-Holland, 1990.

[45] P. D. Mosses. Foundations of modular SOS. In *Proceedings of MFCS'99, 24th International Symposium on Mathematical Foundations of Computer Science*, pages 70–80. Springer LNCS 1672, 1999.

[46] P. D. Mosses. Pragmatics of modular SOS. In *Proceedings of AMAST'02, 9th Intl. Conf. on Algebraic Methodology and Software Technology*, pages 21–40. Springer LNCS 2422, 2002.

[47] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60–61:195–228, 2004.

[48] M. Pettersson. *Compiling Natural Semantics*. Springer Verlag, LNCS 1549, 1999.

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

[50] G. Roşu. Programming language classes. Department of Computer Science, University of Illinois at Urbana-Champaign, `http://fsl.cs.uiuc.edu/~grosu/classes/`.

[51] G. Roşu, R. P. Venkatesan, J. Whittle, and L. Leustean. Certifying optimality of state estimation programs. In *Computer Aided Verification (CAV'03)*, pages 301–314. Springer, 2003. LNCS 2725.

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

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

[54] D. Scott. Outline of a mathematical theory of computation. In *Proceedings, Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176. Princeton University, 1970. Also appeared as Technical Monograph PRG 2, Oxford University, Programming Research Group.

[55] D. Scott and 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.

[56] M.-O. Stehr, I. Cervesato, and S. Reich. An execution environment for the MSR cryptoprotocol specification language. http://formal.cs.uiuc.edu/stehr/msr.html.

[57] M.-O. Stehr and C. Talcott. PLAN in Maude: Specifying an active network programming language. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*, volume 117. ENTCS, Elsevier, 2002.

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

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

[60] D. Turi. *Functorial Operational Semantics and its Denotational Dual*. PhD thesis, Free University, Amsterdam, 1996.

[61] A. Verdejo. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.

[62] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Manuscript, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid, August 2003.

[63] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In *Proc. FORTE/PSTV 2000*, pages 351–366. IFIP, vol. 183, 2000.

[64] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*. ENTCS, Elsevier, 2002.

[65] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.

[66] M. Wand. First-order identities as a defining language. *Acta Informatica*, 14:337–357, 1980.