

Modular Rewriting Semantics in Practice

Christiano Braga

Universidade Federal Fluminense, Niterói, Brazil

José Meseguer

University of Illinois at Urbana-Champaign, USA

Abstract

We present a general method to achieve modularity of semantic definitions of programming languages specified as rewrite theories, so that semantic rules do not have to be redefined in language extensions. We illustrate the practical use of this method by means of two language case studies: two different semantics for CCS, and three different semantics for the GNU bc language.

Key words: Rewriting logic, programming languages semantics, modularity

1 Introduction

From its early stages, rewriting logic has been understood as a semantic framework particularly well suited for defining the mathematical and operational semantics of programming languages [22,27,19]. A semantic definition for a programming language \mathcal{L} takes the form of a rewrite theory $\mathcal{R}_{\mathcal{L}}$. The mathematical semantics of \mathcal{L} is then provided by the initial model $\mathcal{T}_{\mathcal{R}_{\mathcal{L}}}$, and \mathcal{L} 's operational semantics is provided by *deductive inference* in rewriting logic within the theory $\mathcal{R}_{\mathcal{L}}$. That giving a rewriting semantics to a programming language is in practice an easy way to develop executable formal definitions of programming languages, which can then be subjected to different tool-supported formal analysis, is by now a well-established fact [40,2,41,37,36,25,38,8,35,39,14,15].

The rewriting logic semantics of programming languages is related to both algebraic semantics and to structural operational semantics, in the sense of combining and extending both [28]. Since equational logic is a sublogic of rewriting logic, rewriting semantics is a natural generalization of *algebraic semantics* (see, e.g., [42,18,5] for early papers, [31] for the relationship with action semantics, and [17] for a recent textbook) where the semantics of a programming language \mathcal{L} is axiomatized as an equational theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$,

so that \mathcal{L} 's mathematical semantics is given by the initial algebra $T_{\Sigma_{\mathcal{L}}/E_{\mathcal{L}}}$, and its operational semantics is given by equational simplification with the (typically Church-Rosser) equations $E_{\mathcal{L}}$. The point of this generalization is that equational logic is well suited for specifying *deterministic* languages, but ill suited for concurrent language specification. In rewriting logic, deterministic features are described by *equations*, but concurrent ones are instead described by *rewrite rules* with a concurrent transition semantics.

It has also been understood from the early stages [22,27,19], that there is a natural semantic mapping of structural operational semantics (SOS) definitions [34] into rewriting logic. In essence, an SOS rule is mapped to a *conditional* rewrite rule [19,38,39,26,28]. Rewriting logic semantics combines the best features of algebraic semantics and SOS in a generalized framework that adds a crucial distinction between equations and rules (determinism vs. concurrency) missing in each of those two formalisms. Furthermore, the agreement between mathematical and operational semantics is extended to this general setting, taking the form of a completeness theorem for rewriting logic [22,6].

Rewriting logic's distinction between equations and rules is of more than academic interest. The point is that, since rewriting with rules R takes place *modulo* the equations E [22], only the rules R contribute to the size of a system's state space, which can be drastically smaller than if all axioms had been given as rules. This observation, combined with the fact that rewriting logic has several high-performance implementations [1,16,11] and associated formal verification tools [12,20], means that we can use rewriting logic language definitions to obtain practical interpreters and language analysis tools essentially *for free*. For example, in the JavaFAN formal analysis tool [14,15], the semantics of Java and the JVM are defined as rewrite theories in Maude, which are then used to perform formal analysis such as symbolic simulation, search, and LTL model checking of Java programs with a performance that compares favorably with that of other Java analysis tools.

Internal advances within rewriting logic have substantially increased its expressive power for programming language semantics purposes, leading to very succinct and expressive semantic rules, which can be executed in current language implementations. Relevant developments in this regard include:

- expressive typed equational logics, such as membership equational logic (**MEL**) [24];
- executability of rewrite rules with rewrites in conditions under very general assumptions [10];
- fine control of rewriting by allowing certain arguments to be *frozen* [6]; and
- language support for rewriting *strategies*, both at the object-level [1,21], and at the metalevel [13,9].

This paper proposes a further advance, namely a general method for making the rewriting semantics of programming languages *modular*, in the sense that extending a language with new features does not require redefining the

previous semantic rules. Our work is inspired by Peter Mosses' *modular structural operational semantics* (**MSOS**) [32,33,30], and builds upon previous joint work with Hermann Haeusler and Peter Mosses on mapping **MSOS** specifications to rewriting logic [3,2,4]. Our approach has some similarities and some differences compared with the **MSOS** approach. On the one hand, we share with **MSOS** the idea of using *record inheritance* to easily allow adding new semantic entities to a language definition without having to change the semantic rules; this is achieved in our case by associative-commutative matching, a technique that had already been used to make rewrite rules modular and extensible in Maude's object-oriented modules [23]. On the other hand, there are some differences with **MSOS**. First of all, **MSOS** is a substantial and novel semantic framework, extending the original **SOS** framework [34] in highly nontrivial ways. By contrast, we only propose a modular *methodology*, which does not change in any way the rewriting logic framework. A second difference comes from our systematic use of *abstract interfaces*, a theme considerably less developed in **SOS**.

The goals of this paper are quite modest, namely: (1) to explain our methodology and its general principles; and (2) to illustrate its *practical* usefulness by means of concrete case studies. Other important topics are left out, including: (1) a detailed discussion of the relationships between **SOS** and rewriting logic (for which we refer to [19,38,39,28] and to our companion paper [26]); and (2) the precise relationship between **MSOS** and our proposed methodology, which is discussed in detail in [26]. We can however summarize that relationship by stating that there is a semantics-preserving mapping $\tau : \mathcal{S} \mapsto \tau(\mathcal{S})$ transforming an **MSOS** specification \mathcal{S} into a rewrite theory $\tau(\mathcal{S})$ satisfying our modularity requirements. The translation τ could be used in practice to build an execution environment for **MSOS** specifications in a language such as Maude [11].

The structure of the paper is as follows. Section 2 provides prerequisites about **MEL** and the generalized rewriting logic over **MEL** of [6]. Section 3 explains our methodology; and Section 4 discusses two language case studies that we have developed in detail, namely two different modular semantics for CCS [29], and three modular semantics for the GNU bc language. Full executable specifications, as well as sample evaluations of programs in both languages, can be found in <http://formal.cs.uiuc.edu/meseguer/modular>.

2 MEL and Generalized Rewriting Logic

This section gathers basic prerequisites about membership equational logic and generalized rewriting logic. Maude 2.0 [11], the language used to specify our case studies, supports all the logical features of **MEL** and rewriting logic described in this section, with a syntax almost identical to the mathematical notation.

- **Reflexivity.** For each $t \in T_\Sigma(X)$, $\frac{}{(\forall X) t \longrightarrow t}$
- **Equality.** $\frac{(\forall X) u \longrightarrow v \quad E \vdash (\forall X)u = u' \quad E \vdash (\forall X)v = v'}{(\forall X) u' \longrightarrow v'}$
- **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 finite substitution $\theta : X \longrightarrow T_\Sigma(Y)$ with, say, $X = \{x_1, \dots, x_n\}$, and $\theta(x_e) = p_e$, $1 \leq e \leq n$, and for each rule in R of the form,

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

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

$$\frac{(\bigwedge_r (\forall Y) p_{j_r} \longrightarrow p'_{j_r}) \quad (\bigwedge_i (\forall Y) \theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j (\forall Y) \theta(v_j) : s_j) \wedge (\bigwedge_l (\forall Y) \theta(w_l) \longrightarrow \theta(w'_l))}{(\forall Y) \theta(t) \longrightarrow \theta'(t')}$$

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

- **Transitivity**

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

Fig. 1. Deduction rules for rewriting logic.

Membership equational logic (**MEL**) [24] is a typed equational logic in which data are first classified by *kinds* and then further classified by *sorts*, with each kind k having an associated set S_k of *sorts*, so that a datum having a kind but not a sort is understood as an *error* or *undefined* element. Given a **MEL** signature Σ , we write $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ to denote respectively the set of ground Σ -terms of kind k and of Σ -terms of kind k over variables in X , where $X = \{x_1 : k_1, \dots, x_n : k_n\}$ is a set of kinded variables. *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 universally quantified Horn clauses on such atomic formulae. A **MEL theory** is then a pair (Σ, E) with E a set of Σ -sentences.

We present the general version of rewrite theories over **MEL** theories defined in [6]. A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, \phi, R)$ consisting of: (i) a **MEL** theory (Σ, E) ; (ii) a function $\phi : \Sigma \rightarrow \wp_f(\mathbb{N})$ assigning to each function symbol $f : k_1 \dots k_n \rightarrow k$ in Σ a set $\phi(f) \subseteq \{1, \dots, n\}$ of *frozen argument positions*; (iii) a set R of (universally quantified) labeled conditional rewrite rules q having

the general form

$$(\forall X) q: t \rightarrow t' \text{ if } \bigwedge_{i \in I} u_i = u'_i \wedge \bigwedge_{j \in J} v_j : s_j \wedge \bigwedge_{l \in L} w_l \rightarrow w'_l \quad (1)$$

where for appropriate kinds k and k_l in K , $t, t' \in T_\Sigma(X)_k$ and $w_l, w'_l \in T_\Sigma(X)_{k_l}$ for $l \in L$.

The function ϕ specifies which arguments of a function symbol f *cannot be rewritten*, which are called *frozen positions*. Note that if the i^{th} position of f is frozen, then in $f(t_1, \dots, t_n)$ any subterm of t_i becomes also frozen. That is, the freezing idea extends to subterms and in particular to variables. Given two terms t, t' we can then define the sets $\phi(t, t')$ and $\nu(t, t')$ of their frozen (resp. unfrozen) variables (see [6]).

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, a *sequent* of \mathcal{R} is a pair of (universally quantified) terms of the same kind t, t' , denoted $(\forall X)t \rightarrow t'$ with $X = \{x_1 : k_1, \dots, x_n : k_n\}$ a set of kinded variables and $t, t' \in T_\Sigma(X)_k$ for some k . We say that \mathcal{R} *entails* the sequent $(\forall X)t \rightarrow t'$, and write $\mathcal{R} \vdash (\forall X)t \rightarrow t'$, if the sequent $(\forall X)t \rightarrow t'$ can be obtained by means of the inference rules in Figure 1. **Reflexivity**, **Transitivity**, and **Equality** are the usual rules for idle rewrites, concatenation of rewrites, and rewriting modulo the **MEL** theory E . **Congruence** allows rewriting the arguments of a generalized operator, subject to the condition that frozen arguments must stay idle. However, any unfrozen argument can still be *concurrently rewritten*, as expressed by the rewrites in the premise. **Replacement** characterizes the concurrent application of a rewrite rule in its most general form (1). It specifies that for any rewrite rule $q \in R$ and for any (kind-preserving) substitution θ such that the condition of q is satisfied, then it is possible to apply the rewrite q to $\theta(t)$. Moreover, if θ' is a second (kind-preserving) substitution for the variables in X such that θ and θ' coincide on all frozen variables $x \in \phi(t, t')$, while the rewrites $(\forall Y)\theta(x) \rightarrow \theta'(x)$ are provable for the unfrozen variables $x \in \nu(t, t')$, then such nested rewrites can be applied *concurrently* with q .

3 Modular Rewriting Semantics

Modularity is only meaningful in the context of an *incremental* specification, where syntax and corresponding semantic axioms are introduced for groups of related features. We can describe an *incremental* presentation of the syntax of a programming language \mathcal{L} as an indexed family of syntax definitions $\{\mathcal{L}_i\}_{i \in I}$, where the index set I is a *poset* with a top element \top , such that: (i) if $i \leq j$, then $\mathcal{L}_i \subseteq \mathcal{L}_j$, and (ii) $\mathcal{L}_\top = \mathcal{L}$. An *incremental rewriting semantics* for \mathcal{L} is then an indexed family of rewrite theories $\{\mathcal{R}_{\mathcal{L}_i}\}_{i \in I}$, with $\mathcal{R}_{\mathcal{L}_i}$ defining the semantics of the language fragment \mathcal{L}_i . Modularity of the incremental rewriting semantics $\{\mathcal{R}_{\mathcal{L}_i} = (\Sigma_i, E_i, \phi_i, R_i)\}_{i \in I}$ means in essence two things. First of all, it should satisfy the following *monotonicity property*: if $i \leq j$,

then there is a theory inclusion $\mathcal{R}_{\mathcal{L}_i} \subseteq \mathcal{R}_{\mathcal{L}_j}$. This is not easy to achieve in SOS, in the sense that typical SOS definitions are often nonmonotonic (see [30]). However, one can always achieve monotonicity *a posteriori*, once one has reached –perhaps after many changes to the axioms– the definition of the top module $\mathcal{R}_{\mathcal{L}_\top}$. One can then just carve out new submodules of $\mathcal{R}_{\mathcal{L}_\top}$ for each of the language fragments \mathcal{L}_i which will then have the “right” axioms. This is a cheat, as can be recognized by the typical inability to inherit those axioms when a quite different new feature is added in a further extension.

That is why, besides monotonicity, we need the second requirement of *extensibility*. Extensibility means that the rewrite rules have been defined in the most abstract and general way possible, so that when we extend a language with new features the previous rules do not have to be modified, and therefore the extension can be made in a monotonic way. In this way, the semantics of each language feature can be defined *once and for all*, so that we do not have to *retract* earlier semantic definitions in a later language extension. This of course also means that modular semantic definitions can be *reused* on a feature by feature basis, so that a large part (or all) of the semantics of a new language could then be easily defined by just renaming the modules defining its features’ syntax according to the concrete syntax of choice. One is therefore interested in *methods* to develop incremental rewriting semantics definitions of programming languages that are as modular as possible, in both the monotonic and extensible senses.

The method we propose uses pairs, called *configurations*; the first component is the *program text*, and the second a *record* with the different *semantic entities* that change as the program is computed. That is, we organize all the semantic entities associated to a program’s computation in a record data structure. For example, one of the record’s fields may be the *store*, another the *environment* of declarations, and yet another the *traces* left by a concurrent process’ execution. We can specify configurations in Maude with the following membership equational theory (a functional module importing the RECORD module shown later in `protecting` mode, that is, adding no more data (“no junk”) and no new equalities (“no confusion”) to records; the `ctor` keyword indicates a *constructor*):

```
fmod CONF is
  protecting RECORD .
  sorts Program Conf .
  op <_,_> : Program Record -> Conf [ctor] .
endfm
```

The first key modularity technique is *record inheritance*, which is accomplished through pattern matching *modulo* associativity, commutativity, and identity. Features added later to a language may necessitate adding new semantic components to the record; but the axioms of older features can be given once and for all in full generality: they will apply just the same with new components in

the record. Here is the Maude specification of the membership equational theory of records. Note Maude’s convention of identifying kinds with connected components in the subsort inclusion poset, and naming them as equivalence classes of sorts in such components. For example, `[PreRecord]` denotes the kind determined by the connected component `{Field,PreRecord}`.

```
fmod RECORD is
  sorts Index Component Field PreRecord Record Truth .
  subsort Field < PreRecord .
  op tt : -> Truth .
  op null : -> PreRecord [ctor] .
  op _,_ : PreRecord PreRecord -> PreRecord [ctor assoc comm id: null] .
  op _:_ : [Index] [Component] -> [Field] [ctor] .
  op {_} : [PreRecord] -> [Record] [ctor] .
  op duplicated : [PreRecord] -> [Truth] .
  var I : Index . vars C C' : Component . var PR : PreRecord .
  eq duplicated((I : C),(I : C'), PR) = tt .
  cmb {PR} : Record if duplicated(PR) /= tt .
endfm
```

A `Field` is defined as a pair of an `Index` and a `Component`. For any sort, say `Field`, the sort in brackets, say `[Field]`, denotes the corresponding kind containing that sort and any other sorts related to it in the subsort inclusion ordering. Meaningful expressions that cannot be given a sort, e.g., `7/0`, have a kind. For example, illegal index-component pairs will be expressions of kind `[Field]`. A `PreRecord` is a possibly empty (`null`) multiset of fields, formed with the union operator `_,_` which is declared to be *associative* (`assoc`), *commutative* (`comm`) and to have `null` as its *identity* (`id`). Maude will then apply all equations and rules *modulo* such equational axioms [11]. Note the conditional membership (`cmb`) defining a `Record` as an “encapsulated” `PreRecord` with no duplicated fields.

Record inheritance means that we can always consider a record with more fields as a special case of one with fewer fields. For example, a record with an environment component indexed by `env` and a store component indexed by `st` can be viewed as a special case of a record with just the environment component. Matching modulo associativity, commutativity, and identity supports record inheritance, because we can always use an extra variable `PR` of sort `PreRecord` to match *any extra fields the record may have*. For example, a function `get-env` extracting the environment component can be defined by

```
eq get-env({env : E , PR}) = E .
```

and will apply to records with any extra fields that are matched by `PR`.

The second key modularity technique is the systematic use of *abstract interfaces*. That is, the sorts specifying key syntactic and semantic entities (for example, `Program`, `Store`, `Env`) are *abstract sorts* for which we:

- only specify the *abstract functions* manipulating them, that is, a given *sig-*

nature, or *interface*, of abstract sorts and functions; *no axioms* are specified about such functions *at the level of abstract sorts*;

- in a language specification no *concrete* syntactic or semantic sorts are ever identified with abstract sorts: they are always either specified as *subsorts* of corresponding abstract sorts, or mapped to abstract sorts by *coercions*; it is *only at the level of such concrete sorts* that *axioms* about abstract or auxiliary functions are specified.

This means that we *make no a priori ontological commitments* as to the nature of the syntactic or semantic entities. It also means that, since the only commitments ever made happen always at the level of *concrete sorts*, one remains forever free to introduce new meaning and structure in any language extension.

A third modularity technique regards the form of the rules. We require that the rewrite rules in the rewrite theories $\mathcal{R}_{\mathcal{L}_i}$ are semantic rules

$$\langle f(t_1, \dots, t_n), u \rangle \longrightarrow \langle t', u' \rangle \text{ if } C,$$

where f is a language feature, e.g., `if-then-else`, u and u' are record expressions and u contains a variable PR of sort `PreRecord` standing for unspecified additional fields and allowing the rule to match by record inheritance. In addition, the following *information hiding* discipline should be followed in u, u' , and in any record expressions appearing in C : besides basic record syntax, only function symbols appearing in the *abstract interfaces* of some of the record's fields can appear in record expressions; any auxiliary functions defined in concrete sorts of those field's components should never be mentioned. This information hiding makes the rules highly extensible, because the concrete representations of the auxiliary semantic entities can be changed or extended without having to change the rules at all.

The combination of these three techniques can be of great help in making semantic definitions modular and easily extensible. That is, we can develop in this way modular incremental semantic definitions for a language \mathcal{L} as a poset-indexed hierarchy $\{\mathcal{R}_{\mathcal{L}_i} = (\Sigma_i, E_i, \phi_i, R_i)\}_{i \in I}$ of rewrite theory *inclusions*, with the full language definition as the top theory in the hierarchy and with the theory `CONF`—which contains `RECORD`—as the bottom of the hierarchy. By following the methods described above, such a modular definition will then be much more easily extensible than if such methods had not been followed. We illustrate this ease of extensibility by means of two case studies in Section 4.

An important variant of our approach is to choose the MEL sublogic of rewriting logic as the *logical framework* in which to define the semantics of a language. This gives us algebraic semantics as a special case of rewriting logic semantics. Of course, in this case the semantics is no longer given by rewrite rules, but by *conditional equations* of the form,

$$\langle f(t_1, \dots, t_n), u \rangle = \langle t', u' \rangle \text{ if } C.$$

This variant makes our modularity techniques available also for algebraic semantics. In fact, as explained in the Introduction and in [28], the best approach in rewriting logic semantics is to *combine* the equational and the rewriting variants of the modular language specification methodology just described. This is most natural in a rewriting logic context, because of its explicit distinction between equations and rules. It allows us to use the right kind of axiom for each feature: equations for deterministic features, and rules for concurrent ones.

3.1 Controlling Rewrite Steps in Conditions

As illustrated by the CCS example in Section 4.1, sometimes one is interested in giving a quite detailed “small step” semantics for a programming language. In such cases, it becomes important to control the *number of steps* of rewrites in the conditions of a rule. Note that in a rewrite rule

$$Q \longrightarrow Q' \text{ if } P_1 \longrightarrow P'_1 \wedge \dots \wedge P_n \longrightarrow P'_n,$$

because of the **Reflexivity** and **Transitivity** inference rules of rewriting logic (see Figure 1 in Section 2) the rewrites $P_i \longrightarrow P'_i$ in the condition are considerably more general: they can have zero, one, or more steps of rewriting. The point is that, by definition, in rewriting logic *all finitary computations are always derivable as sequents*. Suppose that we want to give a “small step” rewriting semantics to a language so that rewrites in conditions are always *one-step* rewrites. How can we achieve this in a general way? We present a method that will work for any of the rewrite theories specified according to our modular methodology:

- (1) We extend the module `CONF` to a system module (rewrite theory):

```

mod RCONF is extending CONF .
  op {_,_} : [Program] [Record] -> [Conf] [ctor] .
  op [_,_] : [Program] [Record] -> [Conf] [ctor] .
  vars P P' : Program . vars R R' : Record .
  crl [step] : < P , R > => < P' , R' > if { P , R } => [ P' , R' ] .
endm

```

- (2) Each semantic rewrite rule is of the form,

$$\{t, u\} \longrightarrow [t', u'] \text{ if } \{v_1, w_1\} \longrightarrow [v'_1, w'_1] \wedge \dots \wedge \{v_n, w_n\} \longrightarrow [v'_n, w'_n] \wedge C, \quad (2)$$

where $n \geq 0$, and C is a (possibly empty) equational condition involving only equations and memberships.

Note that a rewrite theory \mathcal{R} containing only `RCONF` and rules of the form (2) will be such that any proof of a rewrite

$$\mathcal{R} \vdash \langle v, w \rangle \longrightarrow \langle v', w' \rangle$$

can be expressed (up to the equational equivalence of proofs defined in [6]) as either an application of the **Equality** and **Reflexivity** inference rules, or as repeated applications (no application if $n = 1$) of the **Transitivity** rule (see Section 2) to proofs of rewrites of the form,

$$\langle v, w \rangle = \langle v_0, w_0 \rangle \longrightarrow \langle v_1, w_1 \rangle \longrightarrow \dots \langle v_{n-1}, w_{n-1} \rangle \longrightarrow \langle v_n, w_n \rangle = \langle v', w' \rangle, \quad (3)$$

where each rewrite in the sequence is obtained by application of the **Replacement** inference rule to the **step** rule, and by **Equality**. Of course, any such application of the **step** rule exactly mimics a one-step rewrite with a rule of the form (2) in its condition, so the sequences (3) are the finitary *computations*.

Of course, we have now an intrinsically more expressive way of controlling the number of steps in conditions, so that we are not restricted to specifying conditional rules with one-step rewrite conditions. We can, for example, specify a rule requiring one step in its first condition, one or more steps in its second condition, and zero or more steps in its third condition by a rule (with x a variable of sort **Program**, and y of sort **Record**):

$$\{f(t_1, \dots, t_n), u\} \longrightarrow [t', u'] \text{ if} \\ \{v_1, w_1\} \longrightarrow [v'_1, w'_1] \wedge \{v_2, w_2\} \longrightarrow [x, y] \wedge \langle x, y \rangle \longrightarrow \langle v'_2, w'_2 \rangle \wedge \langle v_3, w_3 \rangle \longrightarrow \langle v'_3, w'_3 \rangle.$$

4 Modular Rewriting Semantics in Practice

The proof of the pudding is in the eating. The practical use of a methodology must be demonstrated by convincing case studies. Here we discuss two examples of quite different flavor: CCS [29], and bc. The GNU bc language is essentially a subset of C well suited for numerical computation that is part of the Linux distribution and therefore is a real, yet medium-sized, language. Another nontrivial case study on the use of our methodology, namely a rewriting logic semantics of Concurrent ML can be found in [7].

4.1 Modular Rewriting Semantics of CCS

We give a straightforward semantics to CCS [29] in our framework. Two features of our specification are worth noting. First, it does not require any extensions to the syntax of CCS processes, whereas a supersort extending the **Process** sort with additional syntax was used in previous rewriting logic specifications of CCS to handle traces [19,41] (we instead deal with traces in the record). Secondly, our semantic rules are very generally extensible. We illustrate this by the extension to a weak transition semantics in which silent “ τ ”

steps are disregarded: no changes whatsoever are needed in the CCS semantic rules, and no extra rules have to be added. By contrast, the traditional CCS treatment requires defining a *new* relation (denoted \Rightarrow) for weak transitions, and the extension in [41], though modular, required extra semantic rules dealing explicitly with the weak equivalence itself (we instead identify equivalent traces equationally in the record). Another extensibility aspect implicit in the form of our rules is that totally new features, such as imperative constructs, could easily be added to CCS without requiring any changes in our semantic rules. This would be clearly impossible with the standard SOS rules (see Section 2.5 in [29]) and with the rules in [41].

The main module specifying the CCS semantics is `CCS-SEMANTICS`. Since we are interested in a small-step semantics, the module `RCONF` from Section 3.1 is imported. There are two fields in the record of semantic entities: a *trace* of actions, with index `tr`, and an *environment*, with index `env`, keeping the context of (possibly recursive) definitions of process names. The trace component uses a list of actions from the module `ACTION-LIST`, whereas the environment component uses sets of process definitions, called contexts, and defined in the module `CCS-CONTEXT`. Our Maude syntax for actions, processes and contexts follows that in [41]. The key module is of course `CCS-SEMANTICS`, which we later extend to `WEAK-CCS-SEMANTICS`.

```
fmod ACTION is
  protecting QID .

  sorts Label Act .
  subsorts Qid < Label < Act .

  op tau : -> Act .
  op ~_ : Label -> Label [prec 10] .

  eq ~ ~ l:Label = l:Label .
endfm

fmod PROCESS is
  protecting ACTION .

  sorts ProcessId Process .
  subsorts Qid < ProcessId < Process .

  op 0 : -> Process .
  op _._ : Act Process -> Process [prec 25] .
  op _+_ : Process Process -> Process [assoc comm prec 35] .
  op _|_ : Process Process -> Process [assoc comm prec 30] .
  op _[_/_] : Process Label Label -> Process [prec 20] .
  op _\_ : Process Label -> Process [prec 20] .
endfm
```

```

fmod CCS-CONTEXT is
  extending PROCESS .
  sort Context .

  op _=def_ : ProcessId Process -> Context [prec 40] .
  op none : -> Context .
  op _&_ : Context Context -> Context [assoc comm id: none prec 42] .
  op dupl : [Context] -> [Bool] .

  var x : ProcessId .
  var p p' : Process .
  var c : Context .

  eq dupl(x =def p & x =def p' & c) = true .
endfm

fmod ACTION-LIST is
  protecting ACTION .

  sort ActList .
  subsort Act < ActList .

  op mt : -> ActList .
  op __ : ActList ActList -> ActList [ctor assoc id: mt] .
endfm

mod CCS-SEMANTICS is
  protecting CCS-CONTEXT .
  including RCONF .
  protecting ACTION-LIST .

  sorts Trace Env .
  subsort Process < Program .
  subsorts Trace Env < Component .

  ops tr env : -> Index .

  op nil : -> Trace .          *** abstract interface
  op _;_ : Trace Act -> Trace .  *** abstract interface
  op def : [ProcessId] [Env] -> [Process] .  *** abstract interface

  op [_] : ActList -> Trace [ctor] .  *** concrete sort coercion
  op {_} : [Context] -> [Env] [ctor] .  *** concrete sort coercion

  vars l m : Label .
  var a : Act .

```

```

vars p p' q q' : Process .
var x : ProcessId .
var t : Trace .
var e : Env .
var c : Context .
vars al al' : ActList .
vars pr pr' pr'' : PreRecord .

mb (tr : t) : Field .
mb (env : e) : Field .
cmb {c} : Env if dupl(c) /= true .

eq [al] ; a = [al a] .
ceq def(x,{x =def p & c}) = p if {x =def p & c} : Env .

*** Prefix
r1 {a . p,{(tr : t), pr}} => [p,{(tr : t ; a) , pr}] .

*** Summation
crl {p + q,{(tr : t), pr}} => [p',{(tr : t ; a), pr'}]
  if {p,{(tr : nil), pr}} => [p',{(tr : nil ; a), pr'}] .

*** Composition
crl {p | q,{(tr : t), pr}} => [p' | q,{(tr : t ; a), pr'}]
  if {p,{(tr : nil), pr}} => [p',{(tr : nil ; a), pr'}] .
crl {p | q,{(tr : t), pr}} => [p' | q',{(tr : t ; tau), pr''}]
  if {p,{(tr : nil), pr}} => [p',{(tr : nil ; l), pr'}] /\
    {q,{(tr : nil), pr'}} => [q',{(tr : nil ; (~ l)), pr''}] .

*** Restriction
crl {p \ l,{(tr : t), pr}} => [(p' \ l),{(tr : t ; a), pr'}]
  if {p,{(tr : nil), pr}} => [p',{(tr : nil ; a), pr'}] /\
    a /= l /\ a /= ~ l .

*** Relabeling
crl {p[m / l],{(tr : t), pr}} => [(p'[m / l]), {(tr : t ; m), pr'}]
  if {p,{(tr : nil),pr}} => [p',{(tr : nil ; l),pr'}] .
crl {p[m / l],{(tr : t), pr}} => [(p'[m / l]),{(tr : t ; (~ m)), pr'}]
  if {p,{(tr : nil),pr}} => [p',{(tr : nil ; (~ l)),pr'}] .
crl {p[m / l],{(tr : t), pr}} => [(p'[m / l]),{(tr : t ; a),pr'}]
  if {p,{(tr : nil), pr}} => [p',{(tr : nil ; a), pr'}] /\
    a /= l /\ a /= ~ l .

*** Definition
crl {x,{(tr : t),(env : e), pr}} => [p',{(tr : t ; a),(env : e), pr'}]
  if p := def(x,e) /\
    {p,{(tr : nil),(env : e), pr}} => [p',{(tr : nil ; a),(env : e), pr'}] .

```

endm

Note the use of the “matching equation” `p := def(x,context)` in the last rule’s condition. In Maude 2.0 one can introduce new variables in an equational condition (written then with syntax `:=` and called a matching equation) provided the lefthand side of the equation is a pattern (a variable or a constructor term) whose new variables are then instantiated by matching the righthand side, after it has been reduced to canonical form.

This example illustrates three key points about our methodology. The first point is the use of record inheritance to make the rules as general and as extensible as possible. Note that in the semantic rules we leave open the possibility that new fields might be added to the record in a language extension of CCS by using variables of sort `PreRecord` for those possible extra fields; also, in conditional rules we leave open the possibility that in a future extension –adding for example imperative features to CCS processes– when computing a subexpression of a process expression some of those extra fields could be modified, something that does not happen in CCS itself.

The second point is that concrete semantic sorts –in this case the sorts `ActList` and `Context`– are never identified with abstract sorts –in this case `Trace` and `Env`. Instead, they are always either specified as *subsorts* of corresponding abstract sorts, or, as in this example, are mapped to abstract sorts by *coercions*. In this case we have used the coercions

```
op  [_] : ActList -> Trace [ctor] .    *** concrete sort coercion
op  {_} : [Context] -> [Env] [ctor] .  *** concrete sort coercion
```

where the use of kinds for the second coercion indicates that it is a *partial* function, since an environment is only well-typed if a process name does not have two different definitions in it. Indeed, this requirement is imposed by a conditional membership, using the auxiliary predicate `dup1` (also a partial function) in its condition.

The third, closely related point is the strict adherence to the information hiding discipline by which any record expressions appearing in semantic rules only use function symbols appearing in the abstract interfaces, so that any auxiliary functions defined in concrete sorts are never mentioned. In this example the abstract interface is provided by the functions

```
op nil : -> Trace .                    *** abstract interface
op _;_ : Trace Act -> Trace .          *** abstract interface
op def : [ProcessId] [Env] -> [Process] . *** abstract interface
```

which are the only functions used in the semantic rules: no functions from the `ACTION-LIST` or `CCS-CONTEXT` modules (where the concrete sorts for semantic entities are defined) are ever mentioned in the rules. Adherence to these principles is key for modular extensibility.

Suppose that now we want to define a *weak transition semantics* for CCS in

which τ steps become unobservable. In CCS theory this is denoted by labeled transitions $P \xrightarrow{a} P'$, allowing several τ steps before or after an observable action a . The extension is almost trivial and totally modular: we only need to introduce a *new constructor* from `ActList` to `Trace` corresponding to these new traces in which τ steps are ignored, and define their trace equivalence and the meaning of the abstract “right-cons” function by means of two simple equations.

```

mod WEAK-CCS-SEMANTICS is
  extending CCS-SEMANTICS .

  op  <_> : ActList -> Trace [ctor] .  *** concrete sort coercion

  vars al al' : ActList .
  var a : Act .

  eq < al tau al' > = < al al' > .
  eq < al > ; a = < al a > .
endm

```

Note that no changes whatsoever are required in the semantic rules. In fact, in the `WEAK-CCS-SEMANTICS` module both the original transition semantics and the weak transition semantics are available: they only depend on the initial state chosen for the record. If we want to compute with a process P according to the original transition semantics, the initial configuration will be of the form $\langle P, \{(\text{tr} : [\text{mt}]), (\text{env} : \mathbf{e})\} \rangle$, for \mathbf{e} the chosen environment of process definitions. Instead, if we want to compute with the weak transition semantics, we will use the initial configuration $\langle P, \{(\text{tr} : \langle \text{mt} \rangle), (\text{env} : \mathbf{e})\} \rangle$. Note that both of these initial configurations are syntactically different from –indeed, will never match– the initial configuration patterns of the general form $\langle p, \{(\text{tr} : \text{nil}), \text{pr}\} \rangle$ that are used in some conditions of the semantic rules: such conditions are totally impervious to changes in the internal representations of the concrete traces or environments, and will furthermore accommodate the addition of any other fields to the record.

All specifications for this example –as well as a collection of CCS process expressions for evaluation in Maude using such specifications and the `search` command– can be found in <http://formal.cs.uiuc.edu/meseguer/modular>.

4.2 Modular Rewriting Semantics of *bc*

In this section we discuss the application of our techniques to the specification of three different semantics for the GNU *bc* language: a standard rewriting semantics, a language extension with annotations for physical units, and an equational semantics. We illustrate the ideas with sample specification fragments. Complete modular specifications for the three semantics of *bc* can be

found in the files `bc-rules.maude`, `units.maude`, and `equational-bc.maude` at <http://formal.cs.uiuc.edu/meseguer/modular>.

The GNU `bc` language is an arbitrary precision calculator language whose syntax can essentially be regarded as a subset of the C language. The following definition of the recursive factorial function illustrates the `bc` syntax for function definitions.

```
define f (x) {
  if (x <= 1) return (1);
  return (f(x-1) * x);
}
```

In our modular rewriting semantics specification of `bc` in `bc-rules.maude`, the record has two main components: an *environment* and a *store*. Therefore, variables in `bc` are simply environment *bindings* of variables identifiers to store *locations*.¹ Applying the information-hiding principles, so that only abstract functions are mentioned in semantic rules, ensures that the actual *details* of: (i) how a new memory location is created; (ii) a location's structure; and (iii) how a location is bound to an identifier in the environment are left *open* by the specification of the semantic rules for memory-related `bc` language constructs. For example, the semantic rules that specify the meaning of assignment and variable evaluation make use of *abstract functions* in the interface of an abstract data type for an *extensible* memory model (this is illustrated later with the `units` extension) based on an environment and a store. The *ontological commitments* (i)–(iii) are only made at the level of *concrete subsorts* below the abstract sorts for stores and environments.

The following Maude rule specifies the evaluation of `SV:SimpleVar = E:Expr` which is an assignment to a non-array variable (`SimpleVar`) of the value resulting from the evaluation of the expression (`E:Expr`).

```
cr1 < SV:SimpleVar = E:Expr ; B:Block , {(env : E:Env),(st : S:Store),
                                          PR:PreRecord } > =>
  < C:Comp , {(env : E:Env),(st : S''':Store),PR''':PreRecord } >
if < E:Expr , {(env : E:Env),(st : S:Store),PR:PreRecord } > =>
  < SV:SVal, {(env : E:Env),(st : S':Store),PR':PreRecord } > /\
  < E':Env , S'':Store > :=
  update-env-st(E:Env, S':Store, SV:SimpleVar, SV:SVal) /\
  < B:Block , {(env : E':Env),(st : S''':Store),PR':PreRecord } > =>
  < C:Comp, {(env : E':Env),(st : S''':Store),PR''':PreRecord } > .
```

After the evaluation of an assignment the environment should remain the same as it was before the evaluation. Therefore the scope that follows the assignment (`B:Block`) should be evaluated within the environment available before the evaluation of the assignment (`E:Env`) extended with a new binding from

¹ For this well-known method of specifying variables in imperative languages see, e.g., Plotkin [34].

the bc variable (`SV:SimpleVar`) to the location that has the value produced from the evaluation of the expression `E:Expr` (`VL:Value`). This process is all encapsulated inside the abstract function `update-env-st`. Additional equations specifying the semantics of `update-env-st` on concrete environment and store representations define the behavior of `update-env-st` on such representations, specifying, for instance, how a new store location should be created.

Another example illustrating the use of abstract functions is the rule for the specification of function definitions, shown next.

```

crl < define F:SimpleVar(P:SimpleVarList){auto A:VarList ; B:Block } ;
                                          Q:Block ,
      { (env : E:Env), PR:PreRecord } > =>
      < C:Comp , {(env : E:Env), PR':PreRecord } >
if E':Env := override(E:Env, F:SimpleVar,
                      lambda(P:SimpleVarList ; A:VarList ; B:Block)) /\
      < Q:Block , {(env : E':Env),PR:PreRecord } > =>
      < C:Comp , {(env : E':Env),PR':PreRecord } > .

```

The rule specifies that the program (`Q:Block`) that follows the definition of the function (`F:SimpleVar`) should be evaluated within the environment available before the function definition extended with a new binding between the function name and a function abstraction (`lambda(P:SimpleVarList ; A:VarList ; B:Block)`) formed by the function's formal parameters (`P:SimpleVarList`), local variables (`A:VarList`) and function body (`B:Block`). The actual construction of the new binding is encapsulated inside the `override` abstract function. Its concrete implementation is given by the concrete environment. Note that the environment (`E:Env`) *before* the function declaration is the same *after* the function declaration since, by definition, there should be no side-effects to the environment, as opposed to the store. To cope with this technicality the program after the function declaration (`Q:Block`) is evaluated, in the rule condition, with the environment extended with the function declaration (`E':Env`).

Yet another example is the semantic rule for function calls. The Maude rule for function calls in bc is shown below.

```

crl < F:SimpleVar ( EL:ExprList ), {(env : E:Env),(st : S:Store),
                                     PR:PreRecord} > =>
      < C:Comp , {(env : E:Env), (st : N':Store),(PR':PreRecord)} >
if eval(EL:ExprList, {(env : E:Env),(st : S:Store),PR:PreRecord}) =>
er(VL:ValueList, {(env : E:Env),(st : S':Store),PR':PreRecord}) /\
lambda(P:SimpleVarList ; A:VarList ; B:Block) :=
      find(E:Env, F:SimpleVar) /\
      < E':Env, S'':Store > := makeActualParams(E:Env, S':Store,
                                               P:SimpleVarList, VL:ValueList) /\
      < E'':Env, N:Store > := initAutoVars(E':Env, S'':Store,
                                          A:VarList) /\
      < B:Block, {(env : E'':Env),(st : N:Store),PR':PreRecord} > =>
      < C:Comp , {(env : E'':Env),(st : N':Store),PR'':PreRecord} > .

```

endm

First the actual parameters (EL:ExprList) are evaluated. Then the abstraction ($\text{lambda(P:SimpleVarList ; A:VarList ; B:Block)}$) bound to the function name (F:SimpleVar) is retrieved from the environment (E:Env). A new environment (E':Env) is created as an extension to E:Env with new bindings for the actual parameters and the local variables. Finally the function body (B:Block) is evaluated within this new environment (E':Env). The processes of creating the new environment with the actual parameters and of initializing local variables are encapsulated within the abstract functions `makeActualParams` and `initAutoVars`.

A key point about the use of abstract sorts and functions in our proposed methodology is that we always remain free to change such concrete representations in a language extension, which could either change the concrete representations of stores and environments, or add new fields to the record, such as a field for input-output: the point is that the semantic rules will not have to be changed. We can illustrate these ideas by means of the extension `units.maude` of our bc semantics in the spirit of [8], so that a bc program can be annotated with information about the physical units (meters, seconds, kilograms, etc.) that it manipulates. In this extension, the concrete representation of stores has to be changed: they now map a location to a *pair*, with first component the concrete numerical value, and second component an abstract *unit value*, indicating which kind of unit the value corresponds to.

The following Maude equations specify how the `update-env-st` abstract function updates the (extended) store with a *pair* of a numerical (rational) value and a unit. There are similar equations for updating the store with plain units and values.

```

--- Update with rational-unit pair.
eq update-env-st(< [ V:Var, L:Loc ] CE:CEnv > ,
  < [ L:Loc, RP:RUPair ] RS:RUStore > , V:Var, RP':RUPair) =
  < < [ V:Var, L:Loc ] CE:CEnv > , < [ L:Loc, RP':RUPair ]
    RS:RUStore > > .
eq update-env-st(< CE:CEnv > , < RS:RUStore > , V:Var, RP:RUPair) =
  < < [ V:Var, newLoc(< CE:CEnv >) ] CE:CEnv > ,
  < [ newLoc(< CE:CEnv >), RP:RUPair ] RS:RUStore > > [owise] .

```

The first equation updates the store ($\langle [L:Loc, RP:RUPair] RS:RUStore \rangle$) with a rational-unit pair ($RP':RUPair$) at the location ($L:Loc$) bound to a variable ($V:Var$) in the environment ($\langle [V:Var, L:Loc] CE:CEnv \rangle$). The case when a variable ($V:Var$) does not appear in the environment ($\langle CE:CEnv \rangle$) is handled by the second equation. In that case, a new location (`newLoc(< CE:CEnv >)`) must be created and bound to the variable in the environment ($\langle [V:Var, \text{newLoc}(\langle CE:CEnv \rangle)] CE:CEnv \rangle$). The rational-unit pair ($RP:RUPair$) is then stored in the new location ($[\text{newLoc}(\langle CE:CEnv \rangle), RP:RUPair]$) and inserted into the store ($\langle [\text{newLoc}(\langle CE:CEnv \rangle), RP:RUPair] RS:RUStore$

>).

To illustrate the use of such an extension, let us consider the evaluation of the bc program

```
< print(x + y) ; , { env : < mt-env > , st : < mt-rs > } >
```

that produces the following output.

```
< << 0,fail >>,{(env : < mt-env >),(st : < mt-rs >),output : 0} >
```

When the bc program `print(x + y) ;` is evaluated in the context of an empty environment (`< mt-env >`) and an empty store (`< mt-st >`) the output is `0` but the resulting unit is `fail`. This is so because the units of `x` and `y` were *not* declared; therefore the unit of the addition is `fail`. Such a declaration is possible using the annotation `assume` (not shown in the example) which would not generate a `fail` unit. The actual result depends on the specified unit algebra, but could be simply the unit of `x`.

The fundamental point about the extension `units.maude` is that the semantic rules do not have to be modified: we only need to add some extra equations specifying how `update-env-st` and the other abstract functions behave on the new concrete representations. There is however one caveat: if, as in [8], we had wanted to *monitor* the execution of programs in the extended language, a nonmodular extension would seem to be required. One interesting possibility is to investigate the use of reflection and strategies in such monitoring, so that the semantic rules remain unchanged, and the monitoring corresponds to a different dimension of modularity, namely a strategy overseeing the application of the standard semantic rules.

A third semantics for GNU bc, specified in `equational-bc.maude`, illustrates a very general point already mentioned, namely, that our methodology applies almost verbatim (with equations now playing the analogous role of rewrite rules) to specifications in the **MEL** sublogic. This means that we get as well a *modular algebraic semantics* variant of our methods. This is of great practical interest in semantic definitions of *deterministic* languages, such as imperative sequential languages like GNU bc, for which the equational semantics is in fact more suitable.

We illustrate the use of semantic equations by means of language fragments for variable assignment and function declaration in bc. By comparing with the analogous semantic rules given earlier, there emerges a clear parallel between the use of rewrites in the conditions of a conditional rule and the corresponding use of *matching equations* in its equational counterpart. For example, in the condition of our earlier semantic rule for variable assignment there were two rewrite conditions and a matching equation. In the conditional equation below each of these three conditions now becomes a matching equation. Note, for example, how the result of evaluating the expression `E:Expr` in the assignment is now extracted by matching the variable `SV:SVa1` in the first matching equation.

```

ceq < SV:SimpleVar = E:Expr ; B:Block , {(env : E:Env),(st : S:Store),
PR:PreRecord} > =
  < C:Comp , {(env : E:Env),(st : S''':Store),PR':PreRecord} >
  if < SV:SVal, {(env : E:Env),(st : S':Store),PR':PreRecord} > :=
    < E:Expr , {(env : E:Env),(st : S:Store),PR:PreRecord } > /\
    < E':Env , S'':Store > :=
update-env-st(E:Env, S':Store, SV:SimpleVar, SV:SVal) /\
  < C:Comp, {(env : E':Env),(st : S''':Store),PR':PreRecord} > :=
  < B:Block , {(env : E':Env),(st : S'':Store),PR':PreRecord} > .

```

Similarly, in a function declaration the evaluation of the program (Q:Block) that follows a function declaration is also specified by a matching equation.

```

ceq < define F:SimpleVar(P:SimpleVarList){auto A:VarList ; B:Block } ;
Q:Block ,
  { (env : E:Env), PR:PreRecord } > =
  < C:Comp , {(env : E:Env), PR':PreRecord } >
  if E':Env := override(E:Env, F:SimpleVar,
    lambda(P:SimpleVarList ; A:VarList ; B:Block)) /\
  < C:Comp , {(env : E':Env),PR':PreRecord } > :=
  < Q:Block , {(env : E':Env),PR:PreRecord } > .

```

The file `equational-bc.maude` with our purely equational modular semantics of `bc` can be found in <http://formal.cs.uiuc.edu/meseguer/modular>. Of course, for concurrent languages like `CCS`, rewriting logic specifications are still the right formalism to use. In general, however, modular semantic definitions will involve *both equations and rules*. For example, if we were to extend `bc` with threads, then defining its sequential part with equations and its concurrent features with rules would be the most natural axiomatization, and also the most state space efficient to perform formal analysis such as breadth first search and model checking.

5 Concluding Remarks

We have proposed general methods for developing modular semantic definitions of programming languages in rewriting logic, and have demonstrated the practical use of our methods by means of two case studies: `CCS` and the `GNU bc` language, both involving nontrivial language extensions.

Three important directions for future research include:

- (1) gaining more extensive experience applying our methods to a wide range of programming languages, and building a library of modular semantic definitions of programming constructs of wide applicability;
- (2) using rewriting semantic definitions of programming languages as the core of new software analysis tools in the spirit of [40,41,37,36,38,8,35,39,14,15],

- adding new formal analysis capabilities in areas such as model checking of abstractions and theorem proving support; and
- (3) extending our methods to a *truly concurrent semantics* to make them particularly well suited for specifying mobile languages.

Acknowledgement

Research supported by ONR Grant N00014-02-1-0715 and NSF Grant CCR-0234524. Braga would like to acknowledge additional support from CNPq under processes 552192/2002-3 and 300294/2003-4; and from PROPP/UFF.

We have benefited much from our collaboration with Hermann Haeusler and Peter Mosses on the **MSOS**-rewriting logic connection; we are particularly grateful to Peter Mosses for his comments and suggestions on these ideas. We have learned much from the work of Feng Chen, Azadeh Farzan, Narciso Martí-Oliet, Grigore Roşu, Koushik Sen, Mark-Oliver Stehr, Carolyn Talcott, Prasanna Thati, and Alberto Verdejo on executable programming language definitions in Maude. Narciso Martí-Oliet, Grigore Roşu, Alberto Verdejo, and the anonymous referees provided also very helpful detailed comments on a draft of this work.

References

- [1] Borovanský, P., C. Kirchner, H. Kirchner and P.-E. Moreau, *ELAN from a rewriting logic point of view*, Theoretical Computer Science **285** (2002), pp. 155–185.
- [2] Braga, C., “Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics,” Ph.D. thesis, Departamento de Informática, Pontifícia Universidade Católica de Rio de Janeiro, Brasil (2001).
- [3] Braga, C., E. H. Haeusler, J. Meseguer and P. D. Mosses, *Maude Action Tool: Using reflection to map action semantics to rewriting logic*, in: T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, Springer LNCS **1816**, 2000, pp. 407–421.
- [4] Braga, C., E. H. Haeusler, J. Meseguer and P. D. Mosses, *Mapping modular SOS to rewriting logic*, in: M. Leuschel, editor, *12th International Workshop, LOPSTR 2002, Madrid, Spain*, Springer LNCS **2664**, 2002, pp. 262–277.
- [5] Broy, M., M. Wirsing and P. Pepper, *On the algebraic definition of programming languages*, ACM Trans. on Prog. Lang. and Systems **9** (1987), pp. 54–99.
- [6] Bruni, R. 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, Springer LNCS **2719**, 2003, pp. 252–266.

- [7] Chalub, F. and C. Braga, *A modular rewriting semantics of CML*, in: R. Lins, C. Braga and F. Chalub, editors, *Proc. 8th Brazilian Symposium on Programming Languages, Niterói, Brazil, May 2004* (2004), pp. 31–45.
- [8] Chen, F., G. Roşu and R. P. Venkatesan, *Rule-based analysis of dimensional safety*, in: *Rewriting Techniques and Applications (RTA'03)*, Springer LNCS **2706**, 2003, pp. 197–207.
- [9] Clavel, M., “Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications,” CSLI Publications, 2000.
- [10] Clavel, M., 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** (2002), pp. 187–243.
- [11] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, *Maude 2.0 Manual*, june 2003, <http://maude.cs.uiuc.edu>.
- [12] Clavel, M., F. Durán, S. Eker and J. Meseguer, *Building equational proving tools by reflection in rewriting logic*, in: *CAFE: An Industrial-Strength Algebraic Formal Method* (2000), <http://maude.cs.uiuc.edu>.
- [13] Clavel, M. and J. Meseguer, *Reflection and strategies in rewriting logic*, in: J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science **4** (1996), <http://www.elsevier.nl/locate/entcs/volume4.html>.
- [14] Farzan, A., F. Cheng, J. Meseguer and G. Roşu, *Formal analysis of Java programs in JavaFAN*, to appear in Proc. CAV'04, Springer LNCS, 2004.
- [15] Farzan, A., J. Meseguer and G. Roşu, *Formal JVM code analysis in JavaFAN*, to appear in Proc. AMAST'04, Springer LNCS, 2004.
- [16] Futatsugi, K. and R. Diaconescu, “CafeOBJ Report,” World Scientific, AMAST Series, 1998.
- [17] Goguen, J. A. and G. Malcolm, “Algebraic Semantics of Imperative Programs,” MIT Press, 1996.
- [18] Goguen, J. A. and K. Parsaye-Ghomi, *Algebraic denotational semantics using parameterized abstract modules*, in: J. Diaz and I. Ramos, editors, *Formalizing Programming Concepts*, Springer-Verlag, 1981 pp. 292–309, LNCS, Volume 107.
- [19] Martí-Oliet, N. and J. Meseguer, *Rewriting logic as a logical and semantic framework*, in: D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic, 2nd. Edition*, Kluwer Academic Publishers, 2002 pp. 1–87, first published as SRI Tech. Report SRI-CSL-93-05, August 1993.
- [20] Martí-Oliet, N. and J. Meseguer, *Rewriting logic: roadmap and bibliography*, Theoretical Computer Science **285** (2002), pp. 121–154.

- [21] Martí-Oliet, N., J. Meseguer and A. Verdejo, *Towards a strategy language for Maude*, this volume.
- [22] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.
- [23] Meseguer, J., *A logical theory of concurrent objects and its realization in the Maude language*, in: G. Agha, P. Wegner and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993 pp. 314–390.
- [24] Meseguer, J., *Membership algebra as a logical framework for equational specification*, in: F. Parisi-Presicce, editor, *Proc. WADT'97* (1998), pp. 18–61.
- [25] Meseguer, J., *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*, IOS Press, 2003 pp. 133–193.
- [26] Meseguer, J. and C. Braga, *Modular rewriting semantics of programming languages*, to appear in Proc. AMAST'04, Springer LNCS, 2004.
- [27] Meseguer, J., K. Futatsugi and T. Winkler, *Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents*, in: *Proceedings of the 1992 International Symposium on New Models for Software Architecture, Tokyo, Japan, November 1992* (1992), pp. 61–106.
- [28] Meseguer, J. 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* (2004), to appear.
- [29] Milner, R., “Communication and Concurrency,” Prentice Hall, 1989.
- [30] Mosses, P. D., *Modular structural operational semantics*, manuscript, September 2003, to appear in *J. Logic and Algebraic Programming*.
- [31] Mosses, P. D., *Unified algebras and action semantics*, in: *Proc. Symp. on Theoretical Aspects of Computer Science, STACS'89* (1989).
- [32] Mosses, P. D., *Foundations of modular SOS*, in: *Proceedings of MFCS'99, 24th International Symposium on Mathematical Foundations of Computer Science* (1999), pp. 70–80.
- [33] Mosses, P. D., *Pragmatics of modular SOS*, in: *Proceedings of AMAST'02, 9th Intl. Conf. on Algebraic Methodology and Software Technology* (2002), pp. 21–40.
- [34] Plotkin, G. D., *A structural approach to operational semantics*, Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University (1981).
- [35] Roşu, G., R. P. Venkatesan, J. Whittle and L. Leustean, *Certifying optimality of state estimation programs*, in: *Computer Aided Verification (CAV'03)*, Springer, 2003 pp. 301–314, INCS 2725.

- [36] Stehr, M.-O. 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* (2002).
- [37] Thati, P., 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* (2002).
- [38] Verdejo, A., “Maude como marco semántico ejecutable,” Ph.D. thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain (2003).
- [39] Verdejo, A. and N. Martí-Oliet, *Executable structural operational semantics in Maude*, manuscript, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid, August 2003.
- [40] Verdejo, A. and N. Martí-Oliet, *Executing and verifying CCS in Maude*, technical Report 99-00, Dto. Sistemas Informáticos y Programación, Universidad Complutense, Madrid; also, <http://maude.cs.uiuc.edu>.
- [41] Verdejo, A. 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* (2002).
- [42] Wand, M., *First-order identities as a defining language*, *Acta Informatica* **14** (1980), pp. 337–357.