Strategies and simulations in a semantic framework [⋆]

Narciso Martí-Oliet, Miguel Palomino, and Alberto Verdejo

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
{*narciso,miguelpt,alberto*}*@sip.ucm.es*

Abstract

By means of several examples of structural operational semantics for a variety of languages, we justify the importance and interest of using the notions of strategies and simulations in the semantic framework provided by rewriting logic and implemented in the Maude metalanguage. On the one hand, we describe a basic strategy language for Maude and show its application to CCS, the ambient calculus, and the parallel functional language Eden. On the other hand, we show how the concept of stuttering simulation can be used inside Maude to show that a stack machine correctly implements the operational semantics of a simple functional language.

Key words: strategies, simulations, rewriting logic, Maude, operational semantics

1 Introduction

A unique point of view, formalism, or abstraction level is not sufficient to represent a system and reason about its behavior. Hence it is essential to move from one formalism to another to be able to specify different aspects and requirements of a system: functional correctness, concurrency, security, fault tolerance, etc. This requires the development of formal methods and tools to achieve *formal interoperability*, so that we can relate in a mathematically rigorous way the different and complementary formalizations of a system. Thus, a key step towards formal interoperability is the study of *logical and semantic*

[⋆] Research supported by Spanish MEC projects MIDAS TIC2003–01000 and DESAFIOS TIN2006–15660–C02–01, and by Comunidad de Madrid program PROMESAS S–0505/TIC/0407.

frameworks where we can define, execute, and interoperate different logics, languages, and formalisms.

One such framework is rewriting logic [25], which provides a *semantic framework* where we can express different languages and models of computation. The initial work in this area is [22], which studies the representation of different *operational semantics* in rewriting logic. [1] Later, we extended the notion of semantic framework to *executable semantic framework* [34,37], obtaining representations for several operational semantics that can be executed in the declarative language *Maude* [6,7], based on rewriting logic; thus, Maude becomes a *metalanguage* to specify different kinds of languages.

In this paper we review how to use Maude to specify operational semantics. We will see that the need to add mechanisms that allow for more control of the execution becomes readily apparent and describe a basic *strategy language* we have proposed for Maude [24,9], which will be made available in the next version of the system; several examples of operational semantics where strategies play an essential role are presented, including CCS [29], the ambient calculus [4], and the parallel functional language Eden [20]. This is an updated presentation of work that previously has been described only in conference papers.

Using two different operational semantics for the same functional language, we also motivate the need to relate semantics that work at different levels of abstraction. The notion of *simulation* is appropriate for this, but we notice that the mismatch in atomicity requires the additional flexibility provided by the notion of *stuttering simulation*, that we also introduced in the context of rewriting logic in previous conference papers [23,10]. Here we use this concept to show that a stack machine correctly implements the operational semantics of a functional language. The motivation and application of the simulation notions to proving properties of operational semantics is a new contribution of this paper, which also appears in [30] (in Spanish).

These two strands of work, on strategies and simulations, are relatively independent of each other, but we treat them together here because both of them take place in the context of the rewriting logic framework and because we want to emphasize their common application to the specification of operational semantics in such framework.

[1] Of course, this work did not take place in isolation and it was influenced both by previous work on algebraic semantics in OBJ (including, among others, [12,17,13]), and by contemporaneous work by other researchers on similar ideas (such as, for example, [26,18,19]).

2 Ingredients of rewriting logic

A system is axiomatized in rewriting logic [25] by a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory describing its set of *states* (the static part) as the algebraic data type $T_{\Sigma/E}$ associated to the initial algebra (Σ, E). The system's *transitions* (the dynamic part) are axiomatized by the *conditional rewrite rules* R which are of the form $l : t \longrightarrow t'$ if *cond*, with l a label, t and t' Σ-terms, possibly with variables, and *cond* a condition involving equations and rewrites. Deduction in the logic corresponds to computation with those transitions. Under reasonable assumptions about E and R, rewrite theories are *executable*; in particular, Maude [6,7] offers support for multiple sorts, subsort relations, operator overloading, and much more. Elan [2,3] and CafeOBJ [11] are other rewriting logic implementations.

To illustrate both these ideas and Maude syntax consider the following example. We have some natural numbers written on a blackboard and we are allowed, at any given time, to replace any two of them by their arithmetic mean. In this case the static part corresponds to the representation of the blackboard and the numbers themselves. To represent the numbers we simply import the predefined module `NAT`, which contains a type `Nat` and operators `0`, `s`, `_+_`, and `_quo_` that represent zero, succesor, addition, and integer division, respectively. As for the blackboard, it can be represented as a (nonempty) *multiset*, or bag, of numbers.

```
mod BLACKBOARD is
  including NAT .
  sort Blackboard .
  subsort Nat < Blackboard .
  op __ : Blackboard Blackboard -> Blackboard [assoc comm] .
  vars M N : Nat .
  rl [play] : M N => (M + N) quo 2 .
endm
```

The `subsort` declaration tells Maude that a single number constitutes a valid representation for the blackboard. Multiset union is represented with empty syntax `__`; note that this operator has two *attributes*, `assoc` and `comm`, so that terms of sort `Blackboard` are considered *modulo* associativity and commutativity (e.g., `s(0) 0` and `0 s(0)` become indistinguishable). The system's dynamics is specified by a single *rule*; the word in brackets after the keyword `rl` is the rule's name and is optional. Note that it is enough to specify the behavior of the two numbers that are going to be erased, without considering the rest of the numbers in the blackboard.

Finally, since it will be used in later sections, let us extend this module with two operations for calculating the maximum and minimum of the numbers on the blackboard, and one operation for removing a given number. All these operations are defined equationally by means of equations introduced by the keyword eq.

```
mod EXT-BLACKBOARD is
  including BLACKBOARD .
  ops max min : Blackboard -> Nat .
  op remove : Nat Blackboard -> Blackboard .
  vars M N : Nat .
  var B : Blackboard .
  eq max(N) = N .
  eq max(N M) = if (N > M) then N else M fi .
  eq max(N M B) = if (N > M) then max(N B) else max(M B) fi .
  *** similar equations for min
  eq remove(N, N B) = B .
  eq remove(N, B) = B [otherwise] .
endm
```

3 Operational semantics

Structural operational semantics for different languages are described by means of inference rules of the form

$$\frac{P_1 \to Q_1 \quad \dots \quad P_n \to Q_n}{P_0 \to Q_0},$$

where P_i and Q_i represent states of the execution, and $P_i \to Q_i$ represents transitions between such states [31]. Such an inference rule can be seen as a *conditional rewrite rule* of the form

$$P_0 \longrightarrow Q_0 \quad \textit{if} \quad P_1 \longrightarrow Q_1 \wedge \dots \wedge P_n \longrightarrow Q_n\,;$$

in this way the operational semantics becomes a rewriting logic specification in Maude. The intuitive idea is that a given expression is rewritten until we obtain the value to which this expression is reduced according to the operational semantics. If this representation of the semantics is executable, then we automatically get an *interpreter prototype* for the language in question; this method is studied in detail and applied in [37,34].

3.1 A simple functional language: Fpl--

To illustrate these ideas let us consider a simple functional language called *Fpl*-- (which is Hennessy's *Fpl* language [14] without function declarations and application[2]) and show what its operational semantics looks like in Maude.

The grammar defining the language (operations, Boolean operations, expressions, and Boolean expressions) is:

$$\begin{aligned}
op &::= + \mid - \mid * \\
bop &::= \mathsf{And} \mid \mathsf{Or} \\
e &::= n \mid x \mid e' \; op \; e'' \mid \mathsf{If}\ be\ \mathsf{Then}\ e'\ \mathsf{Else}\ e'' \mid \mathsf{let}\ x = e'\ \mathsf{in}\ e'' \\
be &::= bx \mid \mathsf{T} \mid \mathsf{F} \mid be' \; bop \; be'' \mid \mathsf{Not}\ be' \mid \mathsf{Equal}(e, e')
\end{aligned}$$

where n is a number, x a numerical variable, and bx a Boolean variable.

A *state* in the operational semantics is a pair $\langle \rho, e \rangle$, where ρ is an environment assigning values to variables and e is an *Fpl*-- expression. A final state is a pair $\langle \rho, v \rangle$, where v is a value, i.e., either a number n or a Boolean constant T or F. The operational semantics in Figure 1 (where we have omitted similar rules defining the relation $\rightarrow_B$ for the evaluation of Boolean expressions) then defines a step in the evaluation of an expression

$$\langle \rho, e \rangle \rightarrow_A \langle \rho', e' \rangle.$$

These steps are repeated until the final value of a given expression is obtained.[3]

The translation to Maude of these operational semantics rules (again, we omit the rules for Boolean expressions) is then straightforward:

```
  rl  [Var] : < rho, x > => < rho, rho(x) > .
  rl  [Op]  : < rho, v op v' > => < rho, Ap(op,v,v') > .
  crl [Op]  : < rho, e op e' > => < rho', e'' op e' >
              if < rho, e > => < rho', e'' > .
  crl [Op]  : < rho, e op e' > => < rho', e op e'' >
```

[2] The representation of the complete *Fpl* language in a way more similar to that in [14] can be found in [37], which also includes the algebraic signature corresponding to the language grammar below.

[3] Here the transitions are presented in the form $\langle \rho, e \rangle \rightarrow_A \langle \rho', e' \rangle$ instead of the form $\rho \vdash e \rightarrow_A e'$ as in Hennessy's book [14] in order to ease the comparison with the stack machine semantics in Section 6.

Var $$\frac{}{\langle\rho, x\rangle \rightarrow_A \langle\rho, \rho(x)\rangle}$$

Op $$\frac{}{\langle\rho, v\ op\ v'\rangle \rightarrow_A \langle\rho, Ap(op, v, v')\rangle}$$

$$\frac{\langle\rho, e\rangle \rightarrow_A \langle\rho', e''\rangle}{\langle\rho, e\ op\ e'\rangle \rightarrow_A \langle\rho', e''\ op\ e'\rangle} \qquad \frac{\langle\rho, e'\rangle \rightarrow_A \langle\rho', e''\rangle}{\langle\rho, e\ op\ e'\rangle \rightarrow_A \langle\rho', e\ op\ e''\rangle}$$

If $$\frac{\langle\rho, be\rangle \rightarrow_B \langle\rho', be'\rangle}{\langle\rho, \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e'\rangle \rightarrow_A \langle\rho', \mathsf{If}\ be'\ \mathsf{Then}\ e\ \mathsf{Else}\ e'\rangle}$$

$$\frac{}{\langle\rho, \mathsf{If\ T\ Then}\ e\ \mathsf{Else}\ e'\rangle \rightarrow_A \langle\rho, e\rangle} \qquad \frac{}{\langle\rho, \mathsf{If\ F\ Then}\ e\ \mathsf{Else}\ e'\rangle \rightarrow_A \langle\rho, e'\rangle}$$

Loc $$\frac{\langle\rho, e\rangle \rightarrow_A \langle\rho', e''\rangle}{\langle\rho, \mathsf{let}\ x = e\ \mathsf{in}\ e'\rangle \rightarrow_A \langle\rho', \mathsf{let}\ x = e''\ \mathsf{in}\ e'\rangle}$$

$$\frac{}{\langle\rho, \mathsf{let}\ x = v\ \mathsf{in}\ e'\rangle \rightarrow_A \langle\rho, e'[v/x]\rangle}$$

Fig. 1. *Fpl*-- operational semantics

```
              if < rho, e' > => < rho', e'' > .
crl [If]    : < rho, If be Then e Else e' > =>
               < rho', If be' Then e Else e' >
              if < rho, be > => < rho', be' > .
rl  [If]    : < rho, If T Then e Else e' > => < rho, e > .
rl  [If]    : < rho, If F Then e Else e' > => < rho, e' > .
crl [Loc]   : < rho, let x = e in e' > =>
                < rho', let x = e'' in e' >
              if < rho, e > => < rho', e'' > .
rl  [Loc]   : < rho, let x = v in e' > => < rho, e'[v / x] > .
```

The substitution operation used in rule [Loc] can be defined equationally as follows:

```
op _[_/_] : Exp Val Var -> Exp .
eq n [v / x] = n .
eq y [v / x] = if x == y then v else y fi .
eq (e1 op e2) [v / x] = (e1 [v / x]) op (e2 [v / x]) .
eq (If be Then e1 Else e2) [v / x] =
    If (be[v / x]) Then (e1[v / x]) Else (e2[v / x]) .
eq (let x = e1 in e2) [v / x] = let x = (e1 [v / x]) in e2 .
ceq (let y = e1 in e2) [v / x] =
     let y = (e1 [v / x]) in (e2 [v / x])
     if x =/= y .
```

Due to the simplicity of the language and the way the left and right-hand sides

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L}\ \alpha \notin L \cup \overline{L} \qquad \frac{P \xrightarrow{\alpha} P'}{X \xrightarrow{\alpha} P'}\ X =_{def} P$$

Fig. 2. CCS operational semantics

of the rules have been represented, they can be used directly as an interpreter prototype. This is not always the case, as we will see below.

We have shown here a (small-step) computation semantics for a functional language. Maude can be used in the same way to represent (big-step) evaluation semantics as well as different semantics for imperative languages [37]. More recent work on operational semantics in Maude includes Meseguer and Roşu's continuation-based approach [28].

3.2 CCS

As another example we present the operational semantics for the process algebra CCS [29]. Its defining rules appear in Figure 2.

First we define the CCS syntax in Maude. Quoted identifiers are used to represent labels and process identifiers. Notice the attributes `assoc` and `comm` for the summation and parallel composition operators. The operators' precedence is set by means of the attribute `prec`.

```
fmod CCS-SYNTAX is
  protecting QID .
  sorts Label Act ProcessId Process .
  subsorts Qid < Label < Act .
  subsorts Qid < ProcessId < Process .
  op ~_ : Label -> Label .
  eq ~ ~ L:Label = L:Label .
  op tau : -> Act .
  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
```

Full CCS is represented, including (possibly recursive) process definitions by means of *contexts*. A context is well-formed if a process identifier is defined at most once (the union of contexts `_&_` is a partial operation, as shown by the arrow `~>`). We use a conditional membership axiom (`cmb`) to establish which terms are well-formed contexts (of sort `Context`). The evaluation of `def(X, C)` returns the process associated to process identifier `X` if it exists; otherwise, such term does not reduce, remaining as an error term [7].

```
fmod CCS-CONTEXT is
  protecting CCS-SYNTAX .
  sort Context .
  op _=def_ : ProcessId Process -> Context [prec 40] .
  op nil : -> Context .
  op _&_ : Context Context ~> Context [assoc comm id: nil prec 42] .
  op _definedIn_ : ProcessId Context -> Bool .
  op def : ProcessId Context ~> Process .
  op context : -> Context .
  var X : ProcessId .
  var P : Process .
  var C : Context .
  cmb (X =def P) & C : Context if not(X definedIn C) .
  eq X definedIn (X =def P & C) = true .
  eq X definedIn C = false [otherwise].
  eq def(X, (X =def P) & C) = P .
endfm
```

This module includes a constant `context` used to keep the definitions of the process identifiers appearing in a CCS specification.

In order to implement the CCS semantics in Maude we want to interpret a CCS transition $P \xrightarrow{\alpha} P'$ as a rewrite. However, rewrites have no labels, which are essential in the CCS semantics; therefore, we instead make the label a part of the resulting term, obtaining in this way a rewrite of the form $P \longrightarrow \{\alpha\}P'$, where $\{\alpha\}P'$ is a value of sort `ActProcess`, a supersort of `Process`. The following module includes the CCS semantics implementation.

```
mod CCS-SEMANTICS is
  protecting CCS-CONTEXT .
  sort ActProcess .  subsort Process < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess .
  vars L M : Label .  var X : ProcessId .    var A : Act .
  vars P P' Q Q' : Process .    var AP : ActProcess .
  rl  [prefix] : A . P => {A}P .
  crl [sum]  : P + Q => {A}P' if P => {A}P' .
  crl [par1] : P | Q => {A}(P' | Q) if P => {A}P' .
  crl [par2] : P | Q => {tau}(P' | Q')
               if P => {L}P'  /\  Q => {~ L}Q' .
  crl [rel1] : P[M / L] => {M}(P'[M / L]) if P => {L}P' .
  crl [rel2] : P[M / L] => {~ M}(P'[M / L]) if P => {~ L}P' .
  crl [rel3] : P[M / L] => {A}(P'[M / L])
               if P => {A}P'  /\  A =/= L /\ A =/= ~ L .
  crl [res]  : P \ L => {A}(P' \ L)
               if P => {A}P'  /\  A =/= L /\ A =/= ~ L .
  crl [def]  : X => {A}P if (X definedIn context) /\
                             def(X,context) => {A}P .
*** transitive closure
  crl [trans] : P => {A}AP if P => {A}Q /\ Q => AP .
endm
```

The first nine rules correspond to the CCS operational semantic rules, while the rule `trans` represents the transitive closure of the CCS transition relation.[4] In Section 5.1 we will present a strategy that specifies how these rules have to be combined.

It is important to realize that there exist fundamental differences between both examples:

- The semantics of the functional language is deterministic and its rules can be applied in several different ways to always reach the same final result.
- The semantics of CCS is *nondeterministic*: depending on the way the rules are applied we usually obtain different results, or *nontermination*.
- Moreover, there are rewrite computations that do not make sense. In the case of CCS, for example, rewriting should only occur *at the top*; this can be achieved using several techniques (like using the `frozen` attribute that forbids rewriting of an operator's arguments), as described in [34,37]. We will see in Section 5.1 how strategies can also help in this respect.

4 Strategies to control rewriting

Since there are no confluence or termination requirements on the rules in a rewrite theory, in many cases the rewriting process has to be controlled so that it does not become "lost." The Maude system provides commands to explore either a single rewriting computation (`rewrite`) or all of them (`search`), but

[4] The rewrites allowed by the rule `trans` include the one-step rewrites allowed by the other rules (applied to solve the first rewrite condition), because the second rewrite condition of rule `trans` can be solved with *no* rewrites when process `Q` and variable `AP` match directly.

many times we would like to have *strategies* to explore only those computations satisfying some constraints. Indeed, the need to use *strategies* to control the rewriting process was recognized from the beginning in the development of rewriting logic and of systems implementing rewriting logic computation. In particular, strategies are an essential part of the Elan system, that provides a basic set of strategies that can be used when writing rewrite rules, so that at the specification level it is not enforced a separation between rules and strategies [2,3].

Such strategies are “above” the computations they control, i.e., they belong to the *metalevel.* The Maude system implements its own metalevel, allowing in this way the *internal* definition of strategies [7]; however, there are many users who do not want to use such powerful techniques at the metalevel. Motivated by this, we have designed and prototyped, in joint work with J. Meseguer, a basic strategy language to be used at the object level [24,9]. This language allows the definition of strategies to control the way in which rewrite rules are applied. We have benefitted from our own previous experience designing strategy languages in Maude, and also from the experience of other languages like Elan [3,2] and Stratego [38,39]. TOM [1] is also a recently proposed language for programming by means of transformations that can be controlled by using strategies.

Our design is based on a strict separation between the rule level and the strategy level. This is achieved by means of *strategy modules* which associate strategies with a given system module.

4.1 Strategy language

In this section we briefly present most of the elements of the strategy language for Maude. For more details, the reader can see the papers [24,9].

4.1.1 Idle and fail

The simplest strategies are the constants `idle` and `fail`. The first always succeeds, but without modifying the term t to which it is applied, while the second always fails.

4.1.2 Basic strategies

A basic strategy `L[S]` instructs Maude to apply a rule (given its label `L` and a substitution `S` providing values for its variables) to a term, in any position. If the rule is conditional, we can use strategies to indicate how the rewrite conditions have to be checked. Thus, `L[S]{E1,...,En}` denotes a basic strategy that tries to apply the rule `L` with the substitution `S` using the strategy expressions `E1`, ..., `En` to check the n rewrite conditions of the conditional rule `L`. To restrict the application of a rewrite rule to the top of a term (without descending into subterms), the operator `top` is provided.

4.1.3 Tests

There are strategies that work as tests to check a property of a term, by means of pattern matching. `amatch T s.t. C` is a strategy that, when applied to a term `T'`, is successful if there exists a subterm of `T'` that matches the pattern `T` (that is, matching is allowed *anywhere* in the state term) and such matching satisfies the condition `C`. Otherwise, it fails. `match T s.t. C` corresponds to matching only at the *top*. The *such that* fragment can be omitted when the condition `C` is simply `true`.

4.1.4 Operators over strategies

Basic strategies are combined so that strategies are applied to execution paths. The first strategy combinators we consider are the typical regular expressions constructions: concatenation, union, and iteration.

⟨*Strat*⟩ ::=	⟨*Strat*⟩ `;` ⟨*Strat*⟩	concatenation
|	⟨*Strat*⟩ `|` ⟨*Strat*⟩	union
|	⟨*Strat*⟩ `*`	iteration (0 or more)
|	⟨*Strat*⟩ `+`	iteration (1 or more)

4.1.5 Conditional strategies

The if-then-else combinator `_?_:_` allows the definition of conditional strategies, such that its first argument is also a strategy. If the first argument is successful, computation continues with the second argument (also a strategy); otherwise, the first strategy is discarded and the strategy given as third argument is applied.

Using the if-then-else combinator, we can define many other useful strategy combinators as derived operations. `E orelse E'` evaluates `E` in a given state; if such evaluation is successful, its results are the final ones, but if it fails, then `E'` is evaluated in the initial state.

```
E orelse E' = E ? idle : E'
```

`not(E)` reverses the result of evaluating `E`, so that `not(E)` fails when `E` is successful and vice versa.

```
    not(E) = E ? fail : idle
```

An interesting use of `not(E)` is the following "normalization" (or "repeat until the end") operation `E !`:

```
    E ! = E * ; not(E)
```

`try(E)` evaluates `E` in a given state; if it is successful, the corresponding result is given, but if it fails, the initial state is returned.

```
    try(E) = E ? idle : idle
```

Evaluation of `test(E)` checks the success/failure result of `E`, but it does not change the given initial state.

```
    test(E) = not(E) ? fail : idle
```

Notice that `test(E)` = `not(not(E))`.

4.1.6 Decomposition strategies

With the previous combinators we cannot force the application of a strategy to a specific subterm of the given initial term. The operator `amatchrew` allows finer control by means of strategies that rewrite subterms of a given term. When the strategy expression `amatchrew T s.t. C by T1 using E1, ..., Tn using En` is applied to a state term `T'`, first a subterm of `T'` that matches `T` and satisfies `C` is selected. Then, the terms `T1,...,Tn` (which must be disjoint subterms of `T`), instantiated appropriately, are rewritten as described by the strategy expressions `E1,...,En`, respectively. The results are combined in `T` and then substituted in `T'`.

The version `matchrew` works in the same way, but performing matching only at the top. The *congruence operators* used in Elan and Stratego [3,39] are special cases of this `matchrew` combinator, as shown in [24].

4.1.7 Recursion

Recursion is achieved by naming a strategy expression and using this name in the expression itself or in other expressions defining related strategies (examples will be shown in the following sections). Moreover, strategy names can also have arguments [9].

4.2 Implementations of the strategy language

Using the mechanisms provided by the Maude system as a *metalanguage*, we have implemented a prototype of the basic strategy language [24]. The metalevel features of Maude allow the definition of operations to work with modules and computations as objects, as is the case with strategies. The prototype works internally with a labelled version of the computation tree obtained by applying a strategy to a term. The prototype is completed with a user interface providing commands to load modules, execute strategy expressions on states, and show results.

After validating the language experimentally and reaching a mature language design, a direct implementation of our strategy language at the C++ level, at which the Maude system itself is implemented, is currently being developed [9]. This will make the language a stable new feature of Maude and will allow a more efficient execution.

4.3 Simple examples

4.3.1 The blackboard

Recall our blackboard example from Section 2. Now assume that our goal is, by applying the single rule of the system, to get at the end the greatest possible number on the blackboard.

Some possible strategies, among others, consist in always taking the two greatest numbers, or the two smallest, or taking the greatest and the smallest, and are respectively specified by the following strategy definitions: [5]

```
sd maxmax := (matchrew B s.t. X := max(B) /\ Y := max(remove(X,B)) by
                B using play[M <- X ; N <- Y] ) ! .
sd minmin := (matchrew B s.t. X := min(B) /\ Y := min(remove(X,B)) by
                B using play[M <- X ; N <- Y] ) ! .
sd maxmin := (matchrew B s.t. X := max(B) /\ Y := min(B) by
                B using play[M <- X ; N <- Y] ) ! .
```

[5] Incidentally, the strategy `minmin` is optimal. This follows from the fact that to maximize the sum of all the elements on the blackboard after a single step, the two smallest numbers have to be chosen.

The keyword sd (from strategy definition) is used to define strategies in a strategy module.

Strategy expressions E can be utilized in a command srew T using E, which rewrites a term T using a strategy expression E. We can compare the above strategies' behavior over the same initial configuration:

```
Maude> (srew 2000 20 2 200 10 50 using maxmin .)
result NzNat :  178
Maude> (srew 2000 20 2 200 10 50 using maxmax .)
result NzNat :  77
Maude> (srew 2000 20 2 200 10 50 using minmin .)
result NzNat :  1057
```

4.3.2 Insertion sort

Let us now show how to write a strategy that implements the insertion sort algorithm. In the module SORTING, an array is represented as a set of pairs (*index*, *value*), and the rule switch simply swaps the values in two positions of the array.

```
mod SORTING is
  protecting NAT .
  sorts Pair PairSet .
  subsort Pair < PairSet .
  op (_,_) : Nat Nat -> Pair .
  op empty : -> PairSet .
  op __ : PairSet PairSet -> PairSet
          [assoc comm id: empty] .
  op length : PairSet -> Nat .
  vars I J V W : Nat .  var PS : PairSet .
  eq length(empty) = 0 .
  eq length((I, V) PS ) = length(PS) + 1 .
  rl [switch] : (J, V) (I, W) => (J, W) (I, V) .
endm
```

The imperative pseudocode for the insertion sort algorithm is shown in Figure 3 (for sorting an array $V[1..N]$).

The strategies insort and insert below rewrite terms of sort PairSet and represent the loops in the algorithm in a recursive way. Both strategies have a natural number as data argument, which represent the indices used by the algorithm. The expression $X - 1$ is represented as sd(X, 1), where sd is the predefined symmetric difference operation in the NAT module (do not confuse it with the keyword sd for strategy definitions). Notice that the conjunction in the inner loop is separated in two conditions, and that the last strategy def-

```
$Y := 2$
while $Y \leq N$ do
    $X := Y$
    while $X > 1 \;\wedge\; V[X-1] > V[X]$ do
        switch $V[X-1]$ and $V[X]$
        $X := X - 1$
    $Y := Y + 1$
```

Fig. 3. Insertion sort

inition is conditional (introduced by the keyword `csd`), with condition `X > 1`.

```
  sd insort(Y) := try(match PS s.t. Y <= length(PS) ;
                      insert(Y) ;
                      insort(Y + 1)) .
  sd insert(1) := idle .
  csd insert(X) := try(amatch (sd(X,1), V) (X, W) s.t. V > W ;
                       switch[J <- sd(X,1) ; I <- X] ;
                       insert(sd(X,1)))
                   if X > 1 .
```

5 Operational semantics with strategies

In this section we first present a strategy for rewriting CCS processes that controls the rewrite rules presented in Section 3.2. Then the representation of a slightly more complex semantics, namely that for the ambient calculus, is described. Finally, a much more complex case study dealing with the semantics of the parallel functional language Eden is summarized.

5.1 CCS semantics

The transitive closure of the CCS transition relation can be defined in a mathematical way by

$$\frac{P \to P' \qquad P' \to^{*} Q}{P \to^{*} Q}$$

where two kinds of transitions are used, $\to$ and $\to^{*}$. When trying to solve the first premise we know that the rules to be used are the ones defining CCS "one-step" transitions, and that these rules should be applied only at the top of a process term. But when both kinds of transitions are represented in Maude, the same rewrite relation is used (`=>`). That is the reason why we need to control which rules are used when solving the rewrite conditions in rule `trans` in the `CCS-SEMANTICS` module (Section 2). The following strategies control that the rules in the operational semantics of CCS are only applied in the intended way.

```
sd ccs := top(prefix) |
          top(sum{ccs}) |
          top(par1{ccs}) |
          top(par2{ccs, ccs}) |
          top(rel1{ccs}) |
          top(rel2{ccs}) |
          top(rel3{ccs}) |
          top(res{ccs}) |
          top(def{ccs}) .
sd refl-trans := idle | top(trans{ccs, refl-trans}) .
```

The use of strategies to control how the rewrite rules representing the semantics are applied presents an advantage over the techniques explored in [37], where the `frozen` attribute and dummy operators had to be used to avoid undesired rewrites. Now, the rewrite rules are more similar to the semantic rules and the control is completely separated from the transition rules.

5.2 Ambient calculus

In the *ambient calculus* [4] an *ambient* is a place limited by a boundary where computations take place. Its contents are a parallel composition of sequential processes and subambients; communication between processes is local, through a blackboard.

The operational semantics for the ambient calculus consists of a set of structural congruence rules and a set of reduction rules, which can be represented in Maude, as detailed in [32]. It gives us some congruence rules for free, due to the congruence metarule of rewriting logic and the possibility of defining some syntax operators as commutative and associative. The rest of the congruence rules are implemented as Maude equations. The reduction rules are represented as rewrite rules in Maude, as we have shown for the CCS case:

```
rl  [RedIn]   : n[in[m] . P | Q] | m[R] => m[n[P | Q] | R] .
rl  [RedOut]  : m[n[out[m] . P | Q] | R] => n[P | Q] | m[R] .
rl  [RedOpen] : open[n] . P | n[Q] => P | Q .
rl  [RedComm] : ((I)P) | < O > => bound(I,O) P .
crl [RedRes]  : new[k : T] P => new[k : T] Q if P => Q .
crl [RedAmb]  : n[P] => n[Q] if P => Q .
crl [RedPar]  : NSP | NSR => Q | NSR if NSP => Q .
```

We have the interleaving of congruence and reduction rules for free as Maude itself interleaves the application of equations with rewrite rules.

However, the reduction relation of the calculus is not a congruence for all the operators. For example, the fact that `P` $\longrightarrow$ `P'` does not imply that `in[m] . P` $\longrightarrow$ `in[m] . P'`. This means that we cannot freely use the rewrite rules, as Maude would apply them anywhere in a term; and we do not want them to be applied after some operators. This is one of the reasons why the definition of a strategy that controls the application of these rules is necessary.

```
sd norep := top(RedIn) | top(RedOut) | top(RedOpen) | top(RedComm) |
            top(RedAmb{norep}) |
            top(RedPar{norep}) |
            top(RedRes{norep}) .
```

The replication operator raises some problems. It is defined by the following congruence rule: $!P \equiv P\ |!P$. We cannot write it as an equation as for the others because none of the orientations is convenient. This has led us to write this congruence rule as two rewrite rules:

```
rl [Rep] : P => rep(P) .
rl [UnRep] : P | ! P => ! P .
```

that have to be applied by means of a strategy, where `rep` is a function that intuitively replicates processes whenever necessary. We want to apply rule `Rep` only when necessary for subsequent interactions; we have proved that each replicated process has to be unfolded twice for each time we want replication, so `rep`$(!P) = P \mid P\ |!P$. Rule `UnRep` deletes isolated unnecessary copies of the replicated process.

The user provides the desired number of semantic reduction steps, thus controlling termination. In this way, the main strategy for executing an ambient calculus process is the following:

```
sd cg(0) := UnRep ! .
sd cg(s(N)) := (top(Rep) ; norep ; cg(N))  orelse  (UnRep !) .
```

5.3 Eden

Eden [20] is a parallel extension of the functional language Haskell. On behalf of parallelism, Eden overrides Haskell's pure lazy approach, combining a non-strict functional application with eager process creation and communication. The operational semantics of Eden [15] is defined by means of a two-level transition system:

- The lower level handles local effects within processes.
  - · Individual thread evolution $\longrightarrow$.
  - · Parallel thread evolution $\xrightarrow{par}_S$.
- The upper level describes the effects global to the whole system, like process creation and data communication.
  - · Parallel process evolution $\overset{par}{\Longrightarrow}$.
  - · Process creation $\xrightarrow{pc}$, communication $\xrightarrow{com}$, scheduling $\xrightarrow{wUnbl}$, $\xrightarrow{bpc}$, etc.
  - · Transitive closures $\overset{pc}{\Longrightarrow}$, $\overset{com}{\Longrightarrow}$, $\overset{wUnbl}{\Longrightarrow}$, $\overset{bpc}{\Longrightarrow}$, etc.
  - · System evolution $\Longrightarrow$ obtained from the above.

The system then evolves globally as follows. The iteration of the scheduling rules and their sequential composition produces a new global rule:

$$\overset{unbl}{\Longrightarrow} = \overset{wUnbl}{\Longrightarrow} ; \overset{deact}{\Longrightarrow} ; \overset{bpc}{\Longrightarrow} ; \overset{pcd}{\Longrightarrow} ; \overset{vComd}{\Longrightarrow} .$$

The global system evolves by applying the global transition rules:

$$\overset{sys}{\Longrightarrow} = \overset{com}{\Longrightarrow} ; \overset{pc}{\Longrightarrow} ; \overset{unbl}{\Longrightarrow} .$$

Finally, each transition step of the system is defined as follows:

$$\Longrightarrow = \overset{par}{\Longrightarrow} ; \overset{sys}{\Longrightarrow} .$$

Thus the definition of the semantics itself imposes an order in the application of the semantic rules and the use of strategies is mandatory. How to represent in Maude all these rules and relations is studied in [16]. Most of the semantic rules are represented as rewrite rules following the same approach used above for CCS or the ambient calculus. The transition relations that are defined as the concatenation or repetition of other relations are defined by means of strategies as shown by the following examples:

```
sd =wUnbl=> := wUnbl ! .
sd =deact=> := deact ! .

sd =unbl=> := =wUnbl=> ; =deact=> ; =bpc=> ; =pcd=> ; =vComd=> .

sd =sys=> := =com=> ; =pc=> ; =unbl=> .

sd ==> := (matchrew S:System by
                S:System using =par=>(ET(S:System))
          ) ; =sys=> .
```

$$\textbf{(parallel)}\ \frac{\{H_p \xrightarrow{par}_S H'_p\}_{\langle p,H_p\rangle \in S}}{S \overset{par}{\Longrightarrow} \{\langle p, H'_p\rangle\}_{\langle p,H_p\rangle \in S}}$$

Fig. 4. Parallel rule for Eden

But there are a few semantic rules that are more abstract in their mathematical formulation so they cannot be directly translated to rewrite rules. They are represented as (usually recursive) strategies that combine other strategies or rewrite rules. For example, the **(parallel)** rule shown in Figure 4 has a variable number of premises, one for each process in the system S. Each premise makes the corresponding process to evolve exactly once through the transition $\xrightarrow{par}_S$. We implement this rule by means of the strategy `=par=>` that applies the strategy `-par->` to each process in a system. This strategy is recursive and it terminates when the rest of the system (represented by the variable `S:System` below) is empty.

```
sd =par=>(VS:VarSet) :=
     (match empty) ? idle :
     (matchrew P:Process S:System by
          P:Process using -par->(inters(P:Process,VS:VarSet)) ,
          S:System using =par=>(VS:VarSet) ) .
```

The strategy `=par=>` receives as argument the variables corresponding to the threads returned by the function $\mathcal{ET}$ (that returns the set of evolvable threads of a system) applied to the whole system. Strategy `-par->` is called with the set of evolvable variables of process `P`, calculated by function `inters`. For all the details about the representation in Maude of the Eden semantics we refer the reader to [16].

We have been able to represent the operational semantics rules at a quite abstract level, independently from factors such as the eagerness degree in the creation of new processes or the speculation degree. We have also been able to extend this representation to include different measures (such as parallelism degree, speculative computing, communications) *without modifying* the semantics rules. This has been made possible by the separation between rules and strategies and also by the use of arguments in the defined strategies.

6 Abstraction, implementation, and simulation

The Maude system includes a *model checker* to prove temporal properties of systems. Model checking is a very appealing verification technique but, as is well-known, suffers from the state explosion problem so that in many cases it is necessary to *abstract* a system in order to obtain another with a small enough number of states amenable to automatic verification. Symmetrically, some times we have to provide more *concrete* details in the specification of a system, for example when *implementing* a specification and showing its correctness. In general, thus, we see that there is a need to have concepts and methods justifying that a system *simulates* another.

6.1 A stack machine for Fpl--

Recall the functional language *Fpl--* introduced in Section 3.1 along with its operational semantics and let us borrow from [14] another operational semantics for it based on an abstract stack machine. A state of the stack machine is a triple

```
< ST, rho, C >
```

where `ST` is a stack of values, `rho` is an environment assigning values to variables, and `C` is a stack of expressions and operators. An initial state is a triple `< empty, rho, e >` whereas a final state is a triple `< v, rho, empty >`. The transition relations defined in [14] were translated to rewriting logic in [36] and appear in Figure 5.

We want to show that the stack machine *implements correctly* the operational semantics summarized in Figure 1. In both semantics the evaluation of a given expression requires the computation of several steps or transitions; therefore, it seems appropriate to study the relationship between the stepwise computations of both semantics. This is a particular example of the general idea of relating an abstract system with a more concrete one: the tools needed to study them in general are presented in the following sections.

6.2 Transition systems and Kripke structures

When reasoning about computational systems, it is usually convenient to abstract from as many details as possible by means of simple mathematical models that can be used to reason about them [23]. For a state-based system we can represent its behavior by means of a *transition system*, which is a pair $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$, where A is a set of states and $\rightarrow_{\mathcal{A}} \subseteq A \times A$ is a binary relation called the transition relation.

A transition system, however, does not include any information about the relevant properties of the system. In order to reason about such properties it is necessary to add information about the atomic properties that hold in each state. For that we use a *Kripke structure* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$, where $(A, \rightarrow_{\mathcal{A}})$ is

- Analysis rules for the stack machine.

```
rl [Opm1] : < ST, rho, e op e' . C > => < ST, rho, e . e' . op . C > .
rl [Opm1] : < ST, rho, be bop be' . C > =>
            < ST, rho, be . be' . bop . C > .
rl [Ifm1] : < ST, rho, If be Then e Else e' . C > =>
            < ST, rho, be . if(e, e') . C > .
rl [Locm1] : < ST, rho, let x = e in e' . C > =>
             < ST, rho, e . < x, e' > . C > .
rl [Notm1] : < ST, rho, Not be . C > => < ST, rho, be . not . C > .
rl [Eqm1] : < ST, rho, Equal(e, e') . C > =>
            < ST, rho, e . e' . equal . C > .
```

- Application rules for the stack machine

```
rl [Opm2] : < v' . v . ST, rho, op . C > =>
            < Ap(op,v,v') . ST, rho, C > .
rl [Opm2] : < bv' . bv . ST, rho, bop . C > =>
            < Ap(bop,bv,bv') . ST, rho, C > .
crl [Varm] : < ST, rho, x . C > => < v . ST, rho, C >
             if v := lookup(rho,x) .
crl [Varm] : < ST, rho, bx . C > => < bv . ST, rho, C >
             if bv := lookup(rho,bx) .
rl [Valm] : < ST, rho, v . C > => < v . ST, rho, C > .
rl [Valm] : < ST, rho, bv . C > => < bv . ST, rho, C > .
rl [Notm2] : < T . ST, rho, not . C > => < F . ST, rho, C > .
rl [Notm2] : < F . ST, rho, not . C > => < T . ST, rho, C > .
crl [Eqm2] : < v . v' . ST, rho, equal . C > =>
             < T . ST, rho, C > if v = v' .
crl [Eqm2] : < v . v' . ST, rho, equal . C > =>
             < F . ST, rho, C > if v =/= v' .
rl [Ifm2] : < T . ST, rho, if(e, e') . C > => < ST, rho, e . C > .
rl [Ifm2] : < F . ST, rho, if(e, e') . C > => < ST, rho, e' . C > .
rl [Locm2] : < v . ST, rho, < x, e > . C > =>
             < ST, (x,v) . rho, e . pop . C > .
rl [Pop] : < ST, (x,v) . rho, pop . C > => < ST, rho, C > .
```

Fig. 5. Semantics rules for the *Fpl--* stack machine

a transition system such that $\to_{\mathcal{A}}$ is a total relation and $L_{\mathcal{A}} : A \to \mathcal{P}(AP)$ is a labelling function associating each state with a set of atomic properties in AP that it satisfies.

For example, the behaviour of a simple periodic system could be represented by means of a transition system with three states, s_0, s_1, and s_2, and transitions $s_i \to s_{(i+1)\%3}$. Now, to distinguish among the different states and to reason about the system, this transition system can be extended to a Kripke structure by making explicit some atomic properties satisfied by the states, say $L(s_0) = \{\texttt{sleeping}\}$, $L(s_1) = \{\texttt{waiting}\}$, and $L(s_2) = \{\texttt{working}\}$. Note that the relevant properties may vary based on the interest at hand; thus, a less precise alternative would be $L(s_0) = L(s_1) = \{\texttt{off}\}$ and $L(s_2) = \{\texttt{on}\}$.

Kripke structures are specified in rewriting logic by a rewrite system $\mathcal{R} = (\Sigma, E, R)$ so that:

- States are the terms $T_{\Sigma/E,k}$ in the equational theory (Σ, E) with a distinguished type k.
- Transitions are defined from the rules in R: a transition consists in applying a rewrite rule to a unique subterm of the source state.
- We add state predicates Π defined by means of equations D in an equational theory $(\Sigma', E \cup D)$ conservatively extending (Σ, E).

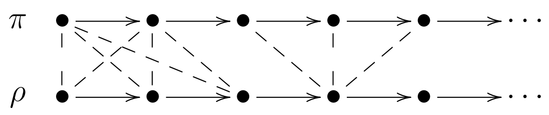
6.3 Simulations

Systems are related using the notion of simulation. Given two transition systems $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$, a *simulation of transition systems* $H : \mathcal{A} \longrightarrow \mathcal{B}$ is a binary relation $H \subseteq A \times B$ such that if $a \rightarrow_{\mathcal{A}} a'$ and aHb then there exists $b' \in B$ with $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$. For two Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ over the same set AP of atomic propositions, an *AP-simulation* furthermore requires that $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$ if aHb. Graphically, the simulation requirement can be represented as follows:

$$\begin{tikzcd}
a \arrow[r, "\mathcal{A}"'] & a' \\
H & H \\
b \arrow[r, "\mathcal{B}"'] & b'
\end{tikzcd}$$

However, this requirement is too strong when we try to relate systems of different granularity, such as our two functional semantics. Intuitively, the stack machine requires many steps to implement a single step of the operational semantics in Section 3.1. What we need is a concept of simulation that allows us to group appropriately several steps.

Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ be transition systems and $H \subseteq A \times B$ a relation. Given a path π in $\mathcal{A}$ and a path ρ in $\mathcal{B}$, we say that ρ *H-matches* π if there are strictly increasing functions $\alpha, \beta : \mathbb{N} \longrightarrow \mathbb{N}$ with $\alpha(0) = \beta(0) = 0$ such that, for all $i, j, k \in \mathbb{N}$, if $\alpha(i) \le j < \alpha(i+1)$ and $\beta(i) \le k < \beta(i+1)$, then $\pi(j)H\rho(k)$. For example, the following diagram shows the beginning of two matching paths, with related elements joined by broken lines and where $\alpha(0) = \beta(0) = 0$, $\alpha(1) = 2$, $\beta(1) = 3$, $\alpha(2) = 5$, etc.



Given two transition systems $\mathcal{A}$ and $\mathcal{B}$, a *stuttering simulation of transition systems* $H : \mathcal{A} \longrightarrow \mathcal{B}$ is a binary relation $H \subseteq A \times B$ such that if aHb then for each path π in $\mathcal{A}$ beginning in a there exists a path ρ in $\mathcal{B}$ beginning in b which H-matches π. Again, for Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ over AP, a *stuttering AP-simulation* $H : \mathcal{A} \longrightarrow \mathcal{B}$ additionally requires that if aHb then $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.

Stuttering simulations resemble the notion of weak (bi)simulation in process algebra, but they do not hide any "actions" from the user. Besides, the latter are defined over labelled transition systems whereas transitions for stuttering simulations are unlabelled, which makes it nontrivial to place them in van Glabbeek's hierarchy [33].

Simulations are important because they *reflect* properties so that we can study the behavior of a system through another one that simulates it. A stuttering AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$ *reflects* the satisfaction of a temporal logic formula $\varphi \in \mathrm{CTL}^*(AP)$ [5] when either

- φ is a state formula and then $\mathcal{B}, b \models \varphi$ and aHb imply $\mathcal{A}, a \models \varphi$; or
- φ is a path formula and then $\mathcal{B}, \rho \models \varphi$ and ρ H-matches π imply $\mathcal{A}, \pi \models \varphi$.

Theorem 1 (reflection theorem) *Stuttering AP-simulations reflect the satisfaction of temporal logic formulas not containing negation or next operators, more specifically, all formulas in the logic* $\mathrm{ACTL}^*\backslash\{\neg, \mathbf{X}\}(AP)$*, see [5].*

PROOF. (Sketch) Given a stuttering AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$, assume that aHb and that ρ H-matches π through α and β. For an atomic proposition p, if $\mathcal{B}, b \models p$ then $p \in L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$, and thus $\mathcal{A}, a \models p$. In the remaining cases, we proceed by induction on the structure of state and path formulas. $\square$

The above definition characterizes stuttering simulations in terms of infinite paths. In [21], an alternative more finitary characterization, called well-founded simulation, is presented, which can also be adapted to our framework and can be easier to work with.

Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ be transition systems. A relation $H \subseteq A \times B$ is a *well-founded simulation of transition systems* from $\mathcal{A}$ to $\mathcal{B}$ if there exist functions $\mu : A \times B \longrightarrow W$ and $\mu' : A \times A \times B \longrightarrow \mathbb{N}$, with $(W, <)$ a well-founded order, such that if aHb and $a \rightarrow_{\mathcal{A}} a'$, then either

- there exists b' such that $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$, or
- $a'Hb$ and $\mu(a', b) < \mu(a, b)$, or
- there exists b' such that $b \rightarrow_{\mathcal{B}} b'$, aHb', and $\mu'(a, a', b') < \mu'(a, a', b)$.

Functions μ and μ' play a role similar to bound functions in the proofs of termination of *while* loops. Notice that when H is a function only the first two conditions are applicable, and in such case the function μ' can be dispensed with. For Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ over AP, a *well-founded AP-simulation* also requires that aHb implies $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.

Theorem 2 *[21] Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ be two Kripke structures over AP and $H \subseteq A \times B$. Then, H is a well-founded AP-simulation if and only if it is a stuttering AP-simulation.*

7 *Fpl--* revisited

We can associate two transition systems, namely, $\mathcal{C} = (C, \rightarrow_{\mathcal{C}})$ and $\mathcal{S} = (S, \rightarrow_{\mathcal{S}})$, to the operational semantics introduced in Section 3.1 and the stack machine described in Section 6.1, respectively. In order to show that the stack machine correctly implements the former operational semantics, we will prove that *there exists a stuttering simulation* of transition systems $h : \mathcal{S} \longrightarrow \mathcal{C}$.

Intuitively, the state `< empty, rho, e >` in S, where `empty` denotes the empty stack of values, should be related with the state `< rho, e >` in C. Now consider, for example, the following derivation for the stack machine:

```
< empty, empty, 2 + 3 > →S < empty, empty, 2 . 3 . + >
                        →S < 2, empty, 3 . + >
                        →S < 3 . 2, empty, + >
                        →S < 5, empty, empty >
```

All the states from the first to the fourth carry the same information, although in different positions; they are obtained by means of analysis rules (see Figure 5). Therefore, it seems appropriate to relate all of them with the same state `< empty, 2 + 3 >`. However, in the fifth state the information has changed and it seems more appropriate to relate this state with `< empty, 5 >`. This last step is an example of an application rule.

So we define $h : \mathcal{S} \longrightarrow \mathcal{C}$ by $h(a) =$ `< rho, e >` if a can be obtained from `< empty, rho, e >` with zero or more applications of the analysis rules for the stack machine together with `Valm` and `Locm2`. Notice that h is a *function*, precisely because not all of the rules can be applied. Moreover, h is *partial*; indeed, it is only defined for reachable states, which form a complete substructure of $\mathcal{S}$ wherein h is total.

Alternatively, by "undoing" the steps taken by the rules, h can be defined by the following set of Maude equations.

```
eq [Base]   : h(< empty, rho, e>) = < rho, e > .
eq [Opm1]   : h(< ST, rho, e . e' . op . C >) =
              h(< ST, rho, e op e' . C >) .
eq [Opm1]   : h(< ST, rho, be . be' . bop . C >) =
              h(< ST, rho, be bop be' . C >) .
eq [Ifm1]   : h(< ST, rho, be . if(e, e') . C >) =
              h(< ST, rho, If be Then e Else e' . C >) .
eq [Locm1] : h(< ST, rho, e. <x, e'> . C >) =
              h(< ST, rho, let x = e in e' . C >) .
eq [Notm1] : h(< ST, rho, be . not. C >) =
              h(< ST, rho, Not be . C >) .
eq [Eqm1]   : h(< ST, rho, e . e' . equal . C >) =
              h(< ST, rho, Equal(e, e') . C >) .
eq [Locm2] : h(< ST, (x, v) . rho, e . pop . C >) =
              h(< v . ST, rho, < x, e > . C >) .
ceq [Valm] : h(< v . ST, rho, C >) = h(< ST, rho, v . C >)
              if not(enabled(C)) .
ceq [Valm] : h(< bv . ST, rho, C >) = h(< ST, rho, bv . C >)
              if not(enabled(C)) .
```

The auxiliary predicate `enabled` used in `Valm` checks that none of the other equations can be applied.

In the proofs that follow we use $e|_p$ to denote the subterm of e at position p, defined in the standard way, and $e[e']_p$ for the term that results from substituting e' for $e|_p$ in e.

Lemma 3 *If $h($`< ST, rho, e . C >`$)$ = `< rho, e' >`, then there exists a position p in* $\mathtt{e}'$ *such that* $\mathtt{e}'|_p = \mathtt{e}$ *and, if* `e` *is not a value, then it is a subexpression that can be reduced in* $\mathtt{e}'$ *with the rules of the first operational semantics (in Section 3.1) in the next step.*

PROOF. Note that the transition relation $\to_{\mathcal{S}}$ is deterministic and that, given a state `< ST, rho, C >`, there is a single way of undoing all the steps to reach a state of the form `< empty, rho, e >`. Therefore, for the purpose of the proof we consider the equations defining h to be oriented rules and proceed by induction on the number of steps used to reach `< rho, e' >`.

When the number of steps is 1 we have $h($`< empty, rho, e >`$) \to$ `< rho, e >` and the result is trivial. Assume that n is greater than 1; we distinguish cases according to the equation (seen as a rule) used for the first step.

- If the first `Opm1` has been applied,

  $h($`< ST, rho, e1 . e2 . op . C >`$) \to h($`< ST, rho, e1 op e2 . C >`$)$.

  By induction hypothesis, there is a position p such that `e'`$|_p$ is `e1 op e2` and then our required position is $p.1$. In addition, since `e1 op e2` is not a value it can be reduced, which implies that `e'` is actually `e1 op e2` and thus `e1` can also be reduced if it is not a value. The same reasoning applies to the other `Opm1`, `Ifm1`, `Notm1`, and `Eqm1`.
- If `Locm1` has been applied,

  $h($`< ST, rho, e1 . < x, e2 > . C >`$)$

  $\to h($`< ST, rho, let x = e1 in e2 . C >`$)$.

  By induction hypothesis, `e'`$|_p$ is `let x = e1 in e2` and we can take $p.1$ as the desired position.
- For `Locm2`,

  $h($`< ST, (x,v) . rho, e . pop . C >`$)$

  $\to h($`< v . ST, rho, < x, e > . C >`$)$

  $\to h($`< ST, rho, v . < x, e > . C >`$)$

  $\to h($`< ST, rho, let x = v in e . C >`$)$.

  By induction hypothesis, `e'`$|_p$ is `let x = v in e` and we can take $p.3$.
- For `Valm`, we have $h($`< v . ST, rho, e . C >`$) \to h($`< ST, rho, v . e . C >`$)$. Now, the only rules that can be applied to the last term are `Opm1` and `Eqm1`; `Valm` is not a valid alternative because it would give rise to three consecutive expressions, which is not possible since there are no ternary operators. Assume that `Eqm1` is used (analogously for the two `Opm1` rules): `C` is of the form `equal . C'` and $h($`< ST, rho, v . e . equal . C' >`$) \to h($`< ST, rho, Equal(v,e) . C' >`$)$. Now, by induction hypothesis, `e'`$|_p$ is `Equal(v,e)` and the required position is $p.2$.

□

Theorem 4 *The partial function $h : \mathcal{S} \longrightarrow \mathcal{C}$ defines a stuttering simulation of transition systems.*

PROOF. We will use the finitary characterization of stuttering simulations given in Theorem 2. Since h is a (partial) function, it is only necessary to define a function $\mu : S \times C \longrightarrow \mathbb{N}$, and we assign to $\mu(a, c)$ the length of the longest path starting at a that only uses analysis rules, `Valm`, or `Locm2`.

Assume that $a \to_{\mathcal{S}} a'$ and that $h(a) = c$. If a' has been obtained by applying an analysis rule, `Valm`, or `Locm2`, then $h(a') = c$ and $\mu(a', c) < \mu(a, c)$. Otherwise, we must find a c' such that $c \to_{\mathcal{C}} c'$ and $h(a') = c'$; we distinguish cases depending on the rule used.

- `Opm2`. In this case, a is `< v' . v . ST, rho, op . C >` and therefore $h(a)$ is equal to $h($`< ST, rho, v op v' . C >`$) =$ `< rho, e >` where, by Lemma 3, there is a position p in `e` such that `e`$|_p$ is `v op v'` and `v op v'` is a subexpression of `e` that can be reduced by the rules of the computation semantics in the next step. We can then take c' to be `< rho, e[Ap(op,v,v')]`$_p$`>`. Similarly for `Notm2`, `Eqm2`, and `Ifm2`.
- `Varm`. Then a must be equal to `< ST, rho, x . C >` and $h(a)$ to `< rho, e >` with `e`$|_p =$ `x` an expression in `e` that can be reduced. Thus, we can take c' to be `< rho, e[rho(x)]`$_p$`>`.
- `Pop`. In this case a must be of the form `< ST, (x,v) . rho, pop . C >`. The only equation that applies to $h(a)$ is `Valm`, and therefore there exists a value `v'` such that `ST` is `v' . ST'`. Applying now the other equations it turns out that $h(a)$ is equal to $h($`< ST', rho, let x = v in v' . C >`$)$, that has to be equal to `< rho, e >` with `e`$|_p =$ `let x = v in v'` a subexpression of `e` that can be reduced. We now take c' to be `< rho, e[v']`$_p$`>`.

Therefore, the conditions of Theorem 2 are satisfied and h is a stuttering simulation of transition systems. $\square$

Notice that h *is not a bisimulation*, i.e., h^{-1} is not a simulation. In the first operational semantics, for a given expression of the form `e op e'` we can choose whether to evaluate `e` before `e'` or the other way around, while the stack machine always evaluates `e` first. That means that, for example, the transition

$$\texttt{< empty, (1 + 2) + (3 + 4) >} \to_{\mathcal{C}} \texttt{< empty, (1 + 2) + 7 >}$$

cannot be simulated by the stack machine.

Furthermore, the simulation h can be lifted to the level of Kripke structures. For that we consider as the set AP of atomic propositions the set of all possible values and extend the transition systems $\mathcal{S}$ and $\mathcal{C}$ with labelling functions $L_{\mathcal{S}}($`< empty, rho, v >`$) = L_{\mathcal{S}}($`< v, rho, empty >`$) = \{$`v`$\}$, $L_{\mathcal{C}}($`< rho,v >`$) = \{$`v`$\}$, and both $L_{\mathcal{S}}(a)$ and $L_{\mathcal{C}}(c)$ are empty otherwise. Applying the reflection theorem, for all expressions `e` and environments `rho` we have

$$\mathcal{C}, \texttt{< rho, e >} \models \mathbf{AF}\texttt{v} \implies \mathcal{S}, \texttt{< empty, rho, e >} \models \mathbf{AF}\texttt{v},$$

where **A** is the universal quantifier over paths and **F** is the temporal operator meaning "eventually in the future." That is, whenever an expression evaluates to a value `v` according to the specification $\mathcal{C}$, then the implementation $\mathcal{S}$ also reaches `v`, so that $\mathcal{S}$ correctly implements $\mathcal{C}$.

8 Summary

The Maude language is very useful to specify executable prototypes for different kinds of systems; in particular, its use as a metalanguage allows the specification of operational semantics for programming and specification languages. However, a strategy language is needed to define in a simpler way the behavior of many systems, as well as the semantics of languages of varying complexities. We have summarized a proposal for such a strategy language which has been used in several case studies, including the operational semantics of the ambient calculus and a quite complex two-level semantics for the parallel functional language Eden. Currently this strategy language is being implemented in C++ at the level of Maude's core rewrite engine [9], and we are also using it with new case studies [35].

Another dimension studied in the context of rewriting logic as a semantic framework is the way to establish relationships between systems, for example, to relate semantics that operate at different levels of atomicity. The concept of stuttering simulation provides a mathematical foundation for this kind of relationship. We have introduced several concepts of simulations and shown some basic results about them; M. Palomino's PhD thesis [30] (in Spanish) deals with this subject in more detail, also covering: equational abstractions, simulations of protocols, algebraic simulations in rewriting logic, and categories of Kripke structures and simulations. Related conference papers in English include [27,23,10]. The study of simulations in the presence of strategies is work in progress. Essentially, no new theoretical foundations are needed: the strategies at hand determine the Kripke structures associated to the rewriting theories and, afterwards, the proofs proceed as usual.

There are still many pending issues in which we are working, or will be. These include proving properties of semantics, automating rule induction, extending the Inductive Theorem Prover (ITP) [8] to be able to deal with rules, the use of reflection with proofs, and the use of strategies along simulations for rewrite systems.

Acknowledgments

We are very grateful to the LSFA'06 organizers for the opportunity to present these results in such an agreeable environment; to José Meseguer for his collaboration in all the different aspects of the research reported here; and to David de Frutos and the anonymous referees for all their insightful comments on previous versions of this paper.

References

[1] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. *TOM Manual*, 2006. `http://tom.loria.fr`.

[2] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[3] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12:69–95, 2001.

[4] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*, LNCS 1387, pages 140–155. Springer, 1998.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[6] 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(2):187–243, 2002.

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

[8] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. Programming and Languages. Special Issue with Extended Versions of Selected Papers from PROLE 2005: The Fifth Spanish Conference on Programming and Languages.

[9] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In *6th International Workshop on Strategies in Automated Deduction (Strategies 2006)*, 2006.

[10] J. L. Fiadeiro, N. Harman, M. Roggenbach, and J. J. M. M. Rutten, editors. *A categorical approach to simulations*, LNCS 3629. Springer, 2005.

[11] K. Futatsugi and R. Diaconescu. *CafeOBJ Report.* World Scientific, AMAST Series, 1998.

[12] J. Goguen, A. Stevens, K. Hobley, and H. Hilberdink. 2OBJ, a metalogical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992.

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

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

[15] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.

[16] M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Using Maude and its strategies for defining a framework for analyzing Eden semantics. In S. Antoy, editor, *The Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS'06*, ENTCS. Elsevier, 2007.

[17] C. Kirchner, H. Kirchner, and A. Mégrelis. OBJ for OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, volume 2 of *Advances in Formal Methods*, chapter 6, pages 307–330. Kluwer Academic Publishers, Boston, 2000.

[18] C. Kirchner, H. Kirchner, and M. Vittek. Implementing computational systems with constraints. In P. Kanellakis, J.-L. Lassez, and V. Saraswat, editors, *Proceedings First Workshop on Principles and Practice of Constraint Programming*, pages 166–175, Brown University, Providence, RI, USA, 1993.

[19] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers*, chapter 8, pages 131–158. MIT Press, 1995.

[20] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(1):431–475, 2005.

[21] P. Manolios. *Mechanical Verification of Reactive Systems.* PhD thesis, University of Texas at Austin, Aug. 2001.

[22] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Technical Report SRI-CSL-93-05, August 1993.

[23] N. Martí-Oliet, J. Meseguer, and M. Palomino. Theoroidal maps as algebraic simulations. In J. L. Fiadeiro, P. Mosses, and F. Orejas, editors, *Recent Trends in Algebraic Development Techniques, 17th International Workshop, WADT*

2004, Barcelona, Spain, March 27-30, 2004, Revised Selected Papers, LNCS 3423, pages 126–143. Springer, 2005.

[24] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, ENTCS 117, pages 417–441. Elsevier, 2005.

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

[26] J. Meseguer, 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*, pages 61–106, Tokyo, Japan, 1992. Research Institute of Software Engineering.

[27] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, LNCS 2741, pages 2–16. Springer, 2003.

[28] J. Meseguer and G. Roşu. The rewriting logic semantics project. In P. Mosses and I. Ulidowski, editors, *Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005), Lisbon, Portugal, 10 July 2005*, LNCS 156(1), pages 27–56. Elsevier, 2006.

[29] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[30] M. Palomino. *Reflexión, abstracción y simulación en la lógica de reescritura.* PhD thesis, Universidad Complutense de Madrid, Spain, Mar. 2005.

[31] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[32] F. Rosa-Velardo, C. Segura, and A. Verdejo. Typed mobile ambients in Maude. In H. Cirstea and N. Martí-Oliet, editors, *Proceedings of the 6th International Workshop on Rule-Based Programming (RULE 2005)*, ENTCS 147, pages 135–161. Elsevier, 2006.

[33] R. J. van Glabbeek. The linear time-branching time spectrum I: The semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of process algebra*, pages 3–99. North-Holland, 2001.

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

[35] A. Verdejo and N. Martí-Oliet. Basic completion by means of Maude strategies. Paper in preparation, 2006.

[36] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. Technical Report 134-03, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2003.

[37] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67:226–293, 2006.

[38] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, LNCS 2051. Springer, 2001.

[39] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer, editor, *Domain-Specific Program Generation*, LNCS 3016, pages 216–238. Springer, 2004.