Electronic Notes in Theoretical Computer Science 71 (2002)
URL: `http://www.elsevier.nl/locate/entcs/volume71.html` 18 pages

Implementing CCS in Maude 2 ⋆

Alberto Verdejo and Narciso Martí-Oliet

Dpto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid. Spain
{alberto,narciso}@sip.ucm.es

Abstract

This paper describes in detail how to bridge the gap between theory and practice in a new implementation of the CCS operational semantics in Maude, where transitions become rewrites and inference rules become conditional rewrite rules with rewrites in the conditions, as made possible by the new features in Maude 2.0. We implement both the usual transition semantics and the weak transition semantics where internal actions are not observed, and on top of them we also implement the Hennessy-Milner modal logic for describing processes. We compare this implementation with a previous one where transitions become judgements and inference rules become rewrites and also comment on extensions to the LOTOS language.

1 Introduction

In the context of proposing rewriting logic as a logical and semantic framework, the paper [9] illustrated several different ways of mapping inference systems into rewriting logic. A very general possibility is to map an inference rule of the form $\frac{S_1 \ldots S_n}{S_0}$ into a rewrite rule of the form $S_1 \ldots S_n \longrightarrow S_0$ that rewrites *multisets* of judgements S_i. This mapping is correct from an abstract point of view, but thinking in terms of executability of the rewrite rules, it is more appropriate to consider rewrites rules of the form $S_0 \longrightarrow S_1 \ldots S_n$ that still rewrite multisets of judgements but go from the conclusion to the premises, so that rewriting with these rewrite rules corresponds to searching for a proof in a bottom-up way. Again this mapping is correct, and in both cases, the intuitive idea is that the rewriting relation corresponds to the horizontal bar separating conclusion from premises in the typical textbook presentation of inference rules.

These mappings can be applied to a wide variety of inference systems, as detailed in [9], including sequent systems for logics and also structural

⋆ Research supported by CICYT project *Desarrollo Formal de Sistemas Basados en Agentes Móviles* (TIC2000–0701–C02–01).

operational semantics definitions for languages. However, in the operational semantics case, judgements S_i typically have the form of some kind of transition $P_i \to Q_i$ between states so that it makes sense to consider the possibility of mapping this transition relation between states to a rewriting relation between terms representing the states. When thinking this way, an inference rule of the form

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

becomes a *conditional* rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \mathit{if} \quad P_1 \longrightarrow Q_1 \wedge \ldots \wedge P_n \longrightarrow Q_n,$$

where the condition includes rewrites.

Rules of this form were already considered by Meseguer in the seminal paper [10] on rewriting logic. At the logical level, this mapping is again correct, but one must be careful to take into account in the mapping additional information appearing in the transitions of the operational semantics. For example, in structural operational semantics for process algebras it is essential for the transitions to have some labelling information that provides the mechanism for synchronization. How to solve these details in the particular case of Milner's CCS [11] was already shown in [9]. Moreover, the papers [3,7] showed the good properties of this semantic mapping for CCS.

With the availability of the first release of the Maude system in 1999 [5], we undertook the project of carefully implementing in a *fully executable* way the CCS operational semantics in order to practically assess the ideas summarized above that theoretically were elegant and correct. The first problem we encountered was that the first Maude release did not allow conditional rules with rewrites in the conditions, that were restricted to Boolean conditions. This did not allow us at that time to consider the approach based on transitions as rewrites, and we were restricted to the approach based on inference rules as rewrites (also known as transitions as judgements in this situation). Even in this case, a number of problems had to be solved, namely the occurrence of new variables in righthand sides of rules, and how to control the nondeterminism in the application of rules. These problems were elegantly solved by means of Maude's reflective capabilities, [1] as described in [16] and summarized below in Section 3. Maude's metalevel features allowed us to bridge some important gaps between theory and practice in a fully executable implementation of CCS operational semantics, and later in some bigger implementations for more complex languages.

Now, the recent availability of implementations for Maude 2.0 [6] have made it possible to reconsider the approach based on transitions as rewrites, because Maude 2.0 allows indeed conditional rules with rewrites in the conditions, where those rewrites are solved at execution time by means of a built-in

[1] Different uses of those reflective capabilities in yet another implementation of CCS as well as in implementing tile systems have also been studied in [4,1].

search mechanism. Thus, we have been able to undertake a continuation of our previous project by carefully reimplementing in a fully *executable* way the CCS operational semantics considering transitions as and using conditional rules with rewrites in the conditions. This paper describes in detail the results obtained in this second implementation and compares both implementations. We advance here that the second implementation is somewhat simpler because it is closer to the mathematical textbook presentation of the operational semantics. However, there is still the need to bridge some gaps between theory and practice, and in this case the new `frozen` attribute available in Maude 2.0 has also played an important role, as described in detail in Section 4.2. The declaration of an operator as frozen forbids rewriting its arguments, thus providing another way of controlling the rewriting process.

Although in this paper we concentrate on CCS operational semantics, the presented techniques are quite general and they are applicable to different languages operational semantics. We have use them to implement a symbolic semantics for LOTOS [15], and in the implementation of the evaluation (big step) and computation (small step) semantics presented by Hennessy in [8].

This paper is organized as follows. After a quick review of CCS syntax and operational semantics in Section 2, we summarize the main ideas of our first executable implementation in Section 3. Section 4 reviews the new Maude 2.0 features that we need in our implementation, and describes in detail how to specify the semantics by means of conditional rules with rewrites in conditions, in a fully executable way. The mechanisms are extended to a weak transition semantics were internal actions are not observed, and then to an implementation of the Hennessy-Milner modal logic for describing processes, which is an essential ingredient of our implementation project. Section 5 compares both implementations, and Section 6 concludes describing some related work, including the extension to LOTOS and an implementation of the semantics in Isabelle.

2 CCS

We give a very brief introduction to Milner's *Calculus of Communicating Systems*, CCS [11]. We assume a set A of *names*; the elements of the set $\overline{A} = \{\overline{a} \mid a \in A\}$ are called *co-names*, and the members of the (disjoint) union $\mathcal{L} = A \cup \overline{A}$ are *labels* naming ordinary actions. The function $a \mapsto \overline{a}$ is extended to $\mathcal{L}$ by defining $\overline{\overline{a}} = a$. There is a special action called *silent action* and denoted τ, intended to represent internal behaviour of a system, and in particular the synchronization of two processes by means of complementary actions a and $\overline{a}$. Then the set of *actions* is $\mathcal{L} \cup \{\tau\}$. The set of processes is intuitively defined as follows:

- 0 is the inactive process that does nothing.
- If α is an action and P is a process, $\alpha.P$ is the process that performs α and

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

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

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

Fig. 1. CCS operational semantics rules

subsequently behaves as P.

- If P and Q are processes, $P + Q$ is the process that may behave as either P or Q.
- If P and Q are processes, $P|Q$ represents P and Q running concurrently with possible communication via synchronization of the pair of ordinary actions a and $\overline{a}$.
- If P is a process and $f : \mathcal{L} \to \mathcal{L}$ is a (finite) relabelling function such that $f(\overline{a}) = \overline{f(a)}$, $P[f]$ is the process that behaves as P but with the actions relabelled according to f, assuming $f(\tau) = \tau$.
- If P is a process and $L \subseteq \mathcal{L}$ is a set of ordinary actions, $P \backslash L$ is the process that behaves as P but with the actions in $L \cup \overline{L}$ prohibited.
- If P is a process, X is a process identifier, and $X =_{def} P$ is a defining equation where P may recursively involve X, then X is a process that behaves as P.

This intuitive explanation can be made precise in terms of the structural operational semantics shown in Figure 1, that defines a labelled transition system for CCS processes. To simplify the presentation, we have already assumed that the operators for summation and parallel composition are commutative and associative, thus using a more abstract syntax.

We define the CCS syntax in Maude, which is nearly common for both implementations of the semantics described in the following sections. The only difference is that the operators for building processes have been defined as *frozen*, a new feature of Maude 2.0 used only in our second implementation. We will explain the reason for this in Section 4.2.

```
fmod CCS-SYNTAX is  inc 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 [frozen prec 25] .
  op _+_ : Process Process -> Process [frozen assoc comm prec 35] .
```

```
  op _|_ : Process Process -> Process [frozen assoc comm prec 30] .
  op _[_/_] : Process Label Label -> Process [frozen prec 20] .
  op _\_ : Process Label -> Process [frozen prec 20] .
endfm
```

We represent full CCS, including (possibly recursive) process definitions by means of *contexts*. We have defined these contexts together with operations to work with them in a module `CCS-CONTEXT` (see details in [16]). It includes a constant `context` used to keep the definitions of the process identifiers used in each CCS specification.

3 Transitions as judgements

We summarize here the main ideas used in our first implementation of the CCS operational semantics where transitions become judgements and inference rules become rewrites [16].

The CCS transition judgement $P \xrightarrow{a} P'$ is represented in Maude by the term P `--` a `->` P' of sort `Judgement`, and the CCS operational semantics rules are translated into rewrite rules where the representation of the conclusion is rewritten to the set of representations of the premises, as the following examples show:

```
crl [res] :  P \ L -- A -> P' \ L
        => ------------------------
                P -- A -> P'       if (A =/= L) and (A =/= ~ L) .

rl [pref] :  A . P -- A -> P
        => --------------------
                emptyJS .
```

In this way, we start with a transition to be proved valid and we work backwards using the rewriting process, maintaining the set of transitions that have to be fulfilled in order to prove the correctness of the initial transition. The initial transition can be rewritten to the empty set if and only if it is a valid transition in the CCS operational semantics.

However, we found two problems while working with this approach. [2] The first one is that sometimes new variables appear in the premises which are not present in the conclusion. Rules of this kind cannot be directly used by the Maude default interpreter; they can only be used at the metalevel using a strategy to instantiate the extra variables. The second problem is that sometimes several rules can be applied to rewrite a judgement, but in general, not all of the possibilities lead to an empty set of judgements. So we have to deal with the whole computation tree of possible rewrites of a judgement, searching if one of the branches leads to `emptyJS`.

In [16] we presented solutions to these problems by modifying the semantics

[2] These problems are intrinsically related to the approach itself and not to the version of Maude used in the implementation.

representation (at the object level) and controlling the rewriting process by means of a strategy at the metalevel.

The presence of new variables was solved by using *explicit metavariables* [13], which make explicit the lack of knowledge that new variables in the righthand side of a rewrite rule represent. The semantics with explicit metavariables has to bind them to concrete values when these values become known. Thus, we introduced in the semantics representation mechanisms to deal with these bindings and propagate them to the rest of judgements where the bound metavariable may be present. The modified representation also has rules with new variables in the righthand side, but now they are localized. The strategy that controls the rewriting process (see below) is in charge of instantiating these variables in order to build *new* metavariables.

The problem of nondeterministic application of rewrite rules was solved by a general search strategy defined at the metalevel. The strategy traverses the conceptual tree of all possible rewrites of a term, built by using the rewrite rules representing the semantics, searching for the term representing the empty set of judgements. If it is found, the transition represented at the root of this tree is a valid CCS transition.

We also extended the semantics implementation by including metavariables as processes (before that, we only needed metavariables as actions). If we start the search strategy with a judgement where the process in the righthand side of the CCS transition is a metavariable, like in `P -- a --> ?P`, and the search reaches the empty set, then the metavariable `?P` has to be bound to one of the one-step successors of the process in the lefthand side, `P`, after performing action `a`. By extending the search strategy to find not only the first way to reach the empty set, but all the possible ways, we implemented a function that returns all the successors of a process after performing a given action. This function was then used to implement the Hennessy-Milner modal logic for CCS processes [14], by following the same techniques for dealing with new variables and with nondeterminism as in the CCS semantics, that is, by defining rewrite rules that rewrite a modal logic judgement $P \models \Phi$ into the set of judgements which have to be satisfied (as specified by the logic semantics) [16]. The search strategy has to be used again, now to check if a modal logic judgement is true. Each time the strategy is used, the module with the rewrite rules that defines the search tree has to be metarepresented. Thus, we obtained three levels of representation. The CCS semantics rules are in the first level. They are controlled by the search strategy at the second level, where the function that returns all the successors of a process and the modal logic semantics are defined. Finally, the modal logic semantics is controlled by the search strategy at the third level.

4 Transitions as rewrites

After reviewing some of the new features of Maude 2.0, we describe the new implementation of the CCS operational semantics where transitions become rewrites and inference rules become conditional rewrite rules.

4.1 Maude 2.0

Maude 2.0 is the new version of Maude, whose key features are: greater generality and expressiveness; efficient support for a wider range of programming applications; and usability as a key component for developing internet programming and mobile computing systems [6]. We briefly summarize here the new features used in the following sections.

Rewrite rules can take the most general possible form in the variant of rewriting logic built on top of membership equational logic. That is, they can be of the form

$$t \to t' \;\; \mathit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \to q_k)$$

with no restriction on which new variables may appear in the righthand side or in the condition. Conditions in rules are formed by an associative conjunction connective `/\`, allowing equations (both ordinary equations `t = t'`, and matching equations `t := t'` where new variables occurring in `t` become instantiated by matching [6]), memberships (`t : s`), and rewrites (`t => t'`) as conditions. In that full generality the execution of a system module will require *strategies* that control at the metalevel the instantiation of the extra variables in the condition and in the righthand side. However, a quite general class of system modules, called *admissible modules*, are executable by Maude 2.0's default interpreter. Essentially, the admissibility requirement ensures that all the extra variables will become instantiated by matching [6].

When executing a conditional rule, the satisfaction of all its conditions is attempted sequentially from left to right; but notice that, besides the fact that many matches for the equational conditions may be possible due to the presence of equational axioms, we also have to deal with the fact that solving rewrite conditions requires *search*, including searching for new solutions when previous ones fail to satisfy subsequent conditions. Therefore, the default interpreter supports search computations. The `search` command looks for all the rewrites of a given term that match a given pattern satisfying some condition (we will see some examples in the following section). Search is reified at the metalevel by means of the function `metaSearch` (used in Section 4.4), which receives as arguments the metarepresented module to work in, the starting term for search, the pattern to search for, a side condition, the kind of search (which may be `'*` for zero or more rewrites, `'+` for one or more rewrites, and `'!` for only matching normal forms), the depth of search, and the required solution number. It returns the term matching the pattern, its type, and the substitution produced by the match.

Another Maude 2.0 feature is the `frozen` attribute. When an operator is declared as frozen, its arguments cannot be rewritten by rules (we will explain why we use this operator in the following section). Note that using this attribute effectively changes the semantics of the frozen operator by disallowing the congruence proof rule.

4.2 Implementation of CCS operational semantics

In order to implement the CCS semantics in Maude with transitions as rewrites, we want to interpret a CCS transition $P \xrightarrow{a} P'$ as a rewriting logic rewrite. However, rewrites have no labels, which are essential in the CCS semantics; therefore, we are going to make the label a part of the resulting term, obtaining in this way a rewrite of the form $P \longrightarrow \{a\}P'$, where $\{a\}P'$ is a value of sort `ActProcess`, a supersort of `Process` (see below what this exactly means in this case). The following module, which is an admissible module [6] and therefore directly executable, includes the CCS semantics implementation.

```
mod CCS-SEMANTICS is  protecting CCS-CONTEXT .
  sort ActProcess .  subsort Process < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess [frozen] .
  vars L M : Label . var A : Act . vars P P' Q Q' : Process . var X : ProcessId .
  *** Prefix
  rl [pref] : A . P => {A}P .
  *** Summation
  crl [sum] : P + Q => {A}P' if P => {A}P' .
  *** Composition
  crl [par] : P | Q => {A}(P' | Q) if P => {A}P' .
  crl [par] : P | Q => {tau}(P' | Q') if P => {L}P' /\ Q => {~ L}Q' .
  *** Relabelling
  crl [rel] : P[M / L] => {M}(P'[M / L]) if P =>{L}P' .
  crl [rel] : P[M / L] => {~ M}(P'[M / L]) if P =>{~ L}P' .
  crl [rel] : P[M / L] => {A}(P'[M / L]) if P =>{A}P' /\ A =/= L /\ A =/= ~ L .
  *** Restriction
  crl [res] : P \ L => {A}(P' \ L) if P => {A}P' /\ A =/= L /\ A =/= ~ L .
  *** Definition
  crl [def] : X => {A}P if (X definedIn context) /\ def(X,context) => {A}P .
endm
```

In this semantic representation, the rewrite rules have the property of being sort-increasing, i.e., in a rewrite $t \longrightarrow t'$, the least sort of t' is bigger than the least sort of t. If we restrict us to terms that can be assigned a sort, and not only a kind, one rule cannot be applied unless the resulting term is well formed, that is, it has a sort. For example, although `A . P` $\longrightarrow$ `{A}P` is a correct transition, we cannot derive `(A . P) | Q` $\longrightarrow$ `({A}P) | Q` because the righthand side term is not well formed. In this way, rewrites are only allowed to happen at the top of a process term, and not inside the term.

But the sort-increasing mechanism is not enough in the current Maude 2.0 system if we have rewrite conditions and infinite processes (those with an infinite number of successors). If we have the rewrite condition `P => {A}Q`, then `P` is tried to be rewritten in any possible way, and the result is matched against the pattern `{A}Q`. If in a concrete application of this rule `P` is `(A . P') | Q'`, it is also rewritten to `({A}P') | Q'` although then the result is rejected. The problem appears when we have recursive processes, because the built-in search that tries to satisfy the rewrite condition can become infinite and not terminate. For example, if `P'` above is recursive, `P' = A . P'`, then `P` is rewritten to `({A}P') | Q'`, `({A}{A}P') | Q'`, etc., although all these results are going to be rejected because they are not well formed. Our solution to this problem has been to declare all the syntax operators as `frozen`, which prevents the arguments of the corresponding operator from being rewritten by rules; see module `CCS-SYNTAX` in Section 2.

However, the problem still appears when we want to know *all* the possible rewrites of the above process `P'` which are of the form `{A}Q` (as we do in Section 4.4 to implement the modal logic semantics). In this case, `P'` is rewritten to `{A}P'`, but also to `{A}{A}P'`, `{A}{A}{A}P'`, etc., and only the first rewrite matches the pattern `{A}Q`. Thus, we have to declare also the operator `{_}_` as frozen. In summary, we use the `frozen` attribute to avoid an infinite loop in the search process when we know that the search would be unsuccessful, although the search may be unsuccessful for two different reasons: either because the built terms are not well formed, as in `({A}P') | Q'`, and that is the reason why the syntax operators are frozen; or because the terms do not match the given pattern, as in `{A}{A}P'`, and that is the reason why `{_}_` is frozen.

A disadvantage is that with the shape of rewrite rules in `CCS-SEMANTICS` and all the constructor operators being declared as frozen, we have lost the ability of proving that a process can perform a sequence of actions, or *trace*, because the rules can only be used to obtain one-step successors. The congruence rule of rewriting logic cannot be used because the operators are frozen, and the transitivity rule cannot be used because all the rules rewrite to something of the form `{A}Q`, and there is no rule with this pattern in the lefthand side. This is not a problem if we want to use the semantics only in the definition of the modal logic semantics, because there only one-step successors are needed.

However, we can solve this by extending the semantics with rules that generate the transitive closure of the CCS transitions as follows:

```
sort TProcess .  subsort TProcess < ActProcess .
op [_] : Process -> TProcess [frozen] .
crl [refl] : [ P ] => {A}Q  if P => {A}Q .
crl [tran] : [ P ] => {A}AP if P => {A}Q /\ [ Q ] => AP .
```

Notice how we use the dummy operator `[_]`. If we did not use it in the lefthand side of the above rules, the lefthand side of both the head of the rule and the rewrites in conditions would be variables that match any term and then the rule itself could be used in order to solve its first condition, giving rise to an infinite loop. In addition, the dummy operator has also been declared as frozen in order to avoid useless rewrites like for example `[ A . P ]` $\longrightarrow$ `[ {A}P ]`.

The obtained representation of CCS, with these two last rules, is semantically correct in the sense that given a CCS process P, there are processes $P_1, \ldots, P_k$ such that

$$P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} P_k$$

if and only if `[ P ]` can be rewritten into `{a1}{a2}...{ak}Pk` (see [9]).

By using the Maude 2.0 `search` command, we can find all the possible one-step successors of a process.

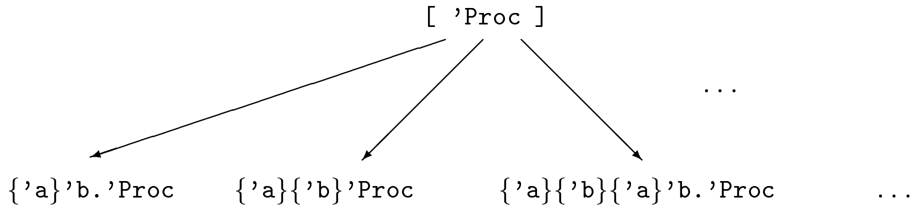
```
Maude> search 'a . 'b . 0 | ~ 'a . 0 => AP:ActProcess .
Solution 1 (state 1)
  AP:ActProcess --> {~ 'a}0 | 'a . 'b . 0
Solution 2 (state 2)
  AP:ActProcess --> {'a}'b . 0 | ~ 'a . 0
Solution 3 (state 3)
  AP:ActProcess --> {tau}0 | 'b . 0
No more solutions.
```

If we add the following equation to the module `CCS-SEMANTICS`, defining the recursive process `'Proc` in the CCS context, we prove that `'Proc` can perform the trace `'a 'b 'a`:

```
  eq context =  'Proc =def 'a . 'b . 'Proc .

Maude> search [1] [ 'Proc ] => {'a}{'b}{'a}X:Process .
Solution 1 (state 5)
  X:Process --> 'b . 'Proc
```

We have asked Maude to search if there is one `[1]` way in which the term `[ 'Proc ]` can be rewritten into the pattern `{'a}{'b}{'a}X:Process`. The `search` command performs a breadth-first search in the conceptual tree of all possible rewrites of term `[ 'Proc ]`, and since there is a solution, it finds it. However, if we ask to search for more solutions, the search does not terminate, because the search tree is infinite.



4.3 Extension to weak transition semantics

Another important transition relation defined in CCS, $P \stackrel{a}{\Longrightarrow} P'$, does not observe τ transitions [11]. It is defined as follows:

$$\frac{P \xrightarrow{\tau}{}^{*}\, Q \quad Q \xrightarrow{a} Q' \quad Q' \xrightarrow{\tau}{}^{*}\, P'}{P \stackrel{a}{\Longrightarrow} P'}$$

where $\xrightarrow{\tau}{}^*$ denotes the reflexive, transitive closure of $\xrightarrow{\tau}$, which is defined in the following way for a general action a

$$\frac{}{P \xrightarrow{a}{}^* P} \qquad\qquad \frac{P \xrightarrow{a} P' \quad P' \xrightarrow{a}{}^* P''}{P \xrightarrow{a}{}^* P''}$$

In our previous work, we implemented this semantics by following the method explained in Section 3, by defining new kinds of judgements and rewrite rules that rewrite the conclusion to the premises. We had to add metavariables as processes since new process variables appear in the premises.

Now we want to implement it following the alternative approach described in Section 4.2 by representing the new transitions also as rewrites, that is, a transition $P \xrightarrow{a}{}^* P$ will be represented as a rewrite $P \longrightarrow \{a\} * P'$ and a transition $P \stackrel{a}{\Longrightarrow} P'$ will be represented as a rewrite $P \longrightarrow \{\{a\}\}P'$. We have to introduce dummy operators again to prevent to use the new rewrite rules in the wrong way in the verification of the rewrite conditions. The proposed implementation is as follows:

```
sorts Act*Process ObsActProcess .
op {_}*_  : Act Process -> Act*Process [frozen] .
op {{_}}_ : Act Process -> ObsActProcess [frozen] .
sort WProcess .  subsorts WProcess < Act*Process ObsActProcess .
op |_| : Process -> WProcess [frozen] .
op <_> : Process -> WProcess [frozen] .
rl  [refl*] : | P | => {tau}* P .
crl [tran*] : | P | => {tau}* R if P => {tau}Q /\ | Q | => {tau}* R .
crl [weak] : < P > => {{A}}P' if | P |  => {tau}* Q  /\
                                   Q    => {A}Q'     /\
                                 | Q' | => {tau}* P' .
```

Notice that both the new semantics operators, `{_}*_` and `{{_}}_`, as well as the dummy operators, `|_|` and `<_>`, are declared as frozen, for the same reasons already explained in Section 4.2.

We can use the `search` command to look for all the weak successors of a given process after performing action `'a`.

```
Maude> search < tau . 'a . tau . 'b . 0 > => {{ 'a }}AP:ActProcess .
Solution 1 (state 2)
  AP:ActProcess --> tau . 'b . 0
Solution 2 (state 3)
  AP:ActProcess --> 'b . 0
No more solutions.
```

4.4 Hennessy-Milner modal logic

We want to implement now the Hennessy-Milner modal logic for describing local capabilities of CCS processes [14]. Formulas are as follows:

$$\Phi ::= \mathtt{tt} \mid \mathtt{ff} \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [K]\Phi \mid \langle K\rangle\Phi \mid [\![K]\!]\Phi \mid \langle\!\langle K\rangle\!\rangle\Phi$$

where K is a set of actions. The satisfaction relation describing when a process P satisfies a property Φ, $P \models \Phi$, is inductively defined in Figure 2.

$$P \models \texttt{tt}$$
$$P \models \Phi_1 \wedge \Phi_2 \text{ iff } P \models \Phi_1 \text{ and } P \models \Phi_2$$
$$P \models \Phi_1 \vee \Phi_2 \text{ iff } P \models \Phi_1 \text{ or } P \models \Phi_2$$
$$P \models [K]\Phi \quad \text{iff } \forall\, Q \in \{P' \mid P \xrightarrow{a} P' \wedge a \in K\}\,.\, Q \models \Phi$$
$$P \models \langle K\rangle\Phi \quad \text{iff } \exists\, Q \in \{P' \mid P \xrightarrow{a} P' \wedge a \in K\}\,.\, Q \models \Phi$$
$$P \models [\![K]\!]\Phi \quad \text{iff } \forall\, Q \in \{P' \mid P \stackrel{a}{\Longrightarrow} P' \wedge a \in K\}\,.\, Q \models \Phi$$
$$P \models \langle\!\langle K\rangle\!\rangle\Phi \quad \text{iff } \exists\, Q \in \{P' \mid P \stackrel{a}{\Longrightarrow} P' \wedge a \in K\}\,.\, Q \models \Phi$$

Fig. 2. Modal logic satisfaction relation.

Since the definition of the satisfaction relation uses the transitions of CCS, we could try to implement it at the same level, with rules like the following ones:

```
rl [and] : P |= Phi /\ Psi => true if P |= Phi => true /\ P |= Psi => true .
rl [dia] : P |= < A > Phi => true if P => {A}Q /\ Q |= Phi .
```

These rules are correct and they exactly represent what the satisfaction relation of the modal logic expresses. For example, the condition of the second rule represents that *there exists* a process Q such that $P \xrightarrow{a} Q$ and $Q \models \Phi$, which is the definition of the diamond modal operator. That is because the variable `Q` is existentially quantified in the rule condition. But we find a problem with the definition of the box modal operator, because it uses a universal quantifier over the possible transitions of a process. If we want to work with all the possible one-step rewrites of a term, we need to go up to the metalevel. By using the operation `metaSearch`, we have defined an operation `succ` that returns all the (metarepresented) successors of a process after performing actions in a given set.

The definition of the operation `succ` in the module `SUCC` below uses two auxiliary functions. The evaluation of `allOneStep(T,N,X)` returns all the one-step rewrites of term `T` (skipping the first `N` solutions) that match the pattern `X` by using rules in module `MOD` (the metarepresentation of `CCS-SEMANTICS`). The evaluation of `filter(F,TS,AS)` returns the metarepresented processes `P` such that the term `F[A,P]` is in `TS` and `A` is in `AS`. In order to look for the term `A` in the term set `AS`, we compare terms in module `MOD`. That is because different, metarepresented terms, like `''a.Qid` and `''a.Act`, can represent the same action in module `CCS-SEMANTICS`. The operation `filter` is used in the definition of `succ(T,TS)` to remove from all the successors of process `T`, those procceses that are reached by performing an action not in the set `TS`.

Having defined these operations in a so general form, we can implement the operation `wsucc` that returns all the weak successors with the same operations.

```
fmod SUCC is  including META-LEVEL .
  op MOD : -> Module .  eq MOD = ['CCS-SEMANTICS] .
  sort TermSet .  subsort Term < TermSet .
```

```
  op mt : -> TermSet .
  op _U_ : TermSet TermSet -> TermSet [assoc comm id: mt] .
  op _isIn_ : Term TermSet -> Bool .
  op allOneStep : Term MachineInt Term -> TermSet .
  op filter : Qid TermSet TermSet -> TermSet .
  op succ : Term TermSet -> TermSet .
  var M : Module .  var F : Qid .  vars T T' X : Term .  var N : MachineInt .
  vars TS AS : TermSet .
  eq T isIn mt = false .
  eq T isIn (T' U TS) =
     (getTerm(metaReduce(MOD, '_==_[T,T'])) == 'true.Bool) or (T isIn TS) .
  eq filter(F, mt, AS) = mt .
  ceq filter(F, X U TS, AS) =
      (if T isIn AS then T' else mt fi) U filter(F, TS, AS) if F[T,T'] := X .
  eq allOneStep(T,N,X) =
     if metaSearch(MOD, T, X, nil, '+, 1, N) == failure then mt
     else  getTerm(metaSearch(MOD, T, X, nil, '+, 1, N)) U
           allOneStep(T, N + 1, X) fi .
  eq succ(T,TS) = filter(('`{_`}_), allOneStep(T, 0, 'AP:ActProcess), TS) .
endfm
```

Using the operation `succ` we have equationally implemented the satisfaction relation of the modal logic. Notice how the semantics for the modal operators is defined by unfolding to a conjunction or disjunction where the successors of the given process are used.

```
fmod MODAL-LOGIC is  protecting SUCC .
  sort HMFormula .
  ops tt ff : -> HMFormula .
  ops _/\_  _\/_ : HMFormula HMFormula -> HMFormula .
  ops <_>_  `[_`]_ : TermSet HMFormula -> HMFormula .
  ops forall exists : TermSet HMFormula -> Bool .
  op _|=_ : Term HMFormula -> Bool .
  var P : Term .  vars K PS : TermSet .  vars Phi Psi : HMFormula .
  eq P |= tt = true .
  eq P |= ff = false .
  eq P |= Phi /\ Psi = P |= Phi and P |= Psi .
  eq P |= Phi \/ Psi = P |= Phi or  P |= Psi .
  eq P |= [ K ] Phi  = forall(succ(P, K), Phi) .
  eq P |= < K > Phi  = exists(succ(P, K), Phi) .
  eq forall(mt, Phi)  = true .
  eq forall(P U PS, Phi) = P |= Phi and forall(PS, Phi) .
  eq exists(mt, Phi)  = false .
  eq exists(P U PS, Phi) = P |= Phi or  exists(PS, Phi) .
endfm
```

Using two examples from [14], we show how we can prove in Maude that a modal formula is satisfied by a CCS process. The first example deals with a vending machine `'Ven` defined in a CCS context as follows:

```
  eq context =  ('Ven    =def  '2p . 'VenB  +  '1p . 'VenL)  &
                ('VenB   =def  'big . 'collectB . 'Ven)      &
                ('VenL   =def  'little . 'collectL . 'Ven) .
```

The process `'Ven` may accept, initially, a `2p` or `1p` coin. If a `2p` coin is deposited, the `big` button may be pressed, and a big item can be collected. If a `1p` coin is deposited, the `little` button may be pressed, and a little item can be collected. After an item is collected, the vending machine goes back to the initial state.

It satisfies that after a coin is deposited and a button is pressed, an item (big or little) can be collected.

```
Maude> red ''Ven.Qid |= [ ''1p.Act + ''2p.Act ] [ ''big.Act + ''little.Act ]
                          < ''collectB.Act + ''collectL.Act > tt .
result Bool: true
```

The second example deals with a railroad crossing system specified as follows:

```
eq context =  ('Road  =def  'car . 'up . ~ 'ccross . ~ 'down . 'Road)   &
   ('Rail   =def  'train . 'green . ~ 'tcross . ~ 'red . 'Rail)        &
   ('Signal =def  ~ 'green . 'red . 'Signal + ~ 'up . 'down . 'Signal)  &
   ('Crossing =def (('Road | ('Rail | 'Signal))
                      \ 'green \ 'red \ 'up \ 'down ))  .
```

The system consists of three components: `Road`, `Rail`, and `Signal`. Actions `car` and `train` represent the approach of a car and a train, `up` opens the gates for the car, $\overline{\texttt{ccross}}$ is the car crossing, `down` closes the gates, `green` is the receipt of a green signal by the train, $\overline{\texttt{tcross}}$ is the train crossing, and `red` sets the light red.

The process `'Crossing` satisfies that when a car and a train arrive to the crossing, exactly one of them has the possibility to cross it.

```
Maude> red ''Crossing.Qid |= [[ ''car.Act ]] [[ ''train.Act ]]
           ((<< '~_[''ccross.Act] >> tt) \/ (<< '~_[''tcross.Act] >> tt)) .
result Bool: true
Maude> red ''Crossing.Qid |= [[ ''car.Act ]] [[ ''train.Act ]]
           ((<< '~_[''ccross.Act] >> tt) /\ (<< '~_[''tcross.Act] >> tt)) .
result Bool: false
```

Maude 2.0 takes 0.5 seconds in solving the last command. With the first implementation in Maude, it takes 10 minutes. The profit is considerable. We compare these two implementations in the following section.

5 Comparison of both approaches

We think that the second implementation has several advantages. This implementation is closer to the mathematical, logical presentation of the semantics. An operational semantics rule establishes that the transition in the conclusion is possible if the transitions in the premises are possible, and that is precisely the interpretation of a conditional rewrite rule with rewrite conditions. The first approach needs auxiliary structures like the multisets of judgements to be proved valid and mechanisms like the generation of new metavariables and their propagation when their concrete values become known. This forced us to implement at the metalevel a search strategy that checks if a given multiset can be reduced to the empty set and generates new metavariables each time they are needed. It is the necessity of new metavariables what makes the strategy unavoidable.

Even if we used Maude 2.0 with the previous semantics implementation, we could not use the `search` command of Maude 2.0, because it cannot handle rewrite rules with new variables in the righthand side whenever they are not bound in any of the conditions, and that is what happens in the first implementation [16]. In the second implementation the necessity of searching appears in the rewrite conditions but the Maude 2.0 system solves the problem, because it is able to handle these conditions together with new variables bound in some condition.

There are also differences found in the things that are done at the object level (level of the semantics representation) and at the metalevel (by using reflection). In the first implementation, the search strategy traverses the conceptual tree with all the possible rewrites of a term, moving continuously between the object level and the metalevel. In the implementation described in this paper, the search occurs completely at the object level, which makes it quite faster (and simpler).

6 Related work

Following the two approaches described in this paper, we have also implemented a symbolic semantics for Full LOTOS [2], which is considerably more complicated than the CCS one. In the first implementation, where transitions are represented as judgements, we improved the ideas presented in Section 3 in several directions, specially regarding efficiency. We changed the sets of judgements to *sequences* of judgements in order to avoid multiple matchings modulo commutativity. This means that judgements will be ordered, and they will be proved from left to right. We also changed the search strategy, in order to try to rewrite each time only the first judgement. Since it has to check that all the judgements can be rewritten to the empty sequence, if the first judgement cannot be rewritten, it does not need to rewrite the rest of judgements, and it can drop all the sequence. Of course, this may affect the way premises are written in a semantic rule; the reduction of a judgement should not require bindings produced by a later (on the right) judgement. The second implementation [15], where transitions are represented as rewrites, does not need improvements regarding efficiency.

In the LOTOS symbolic semantics, the concept of syntactic substitution is used, where a LOTOS variable or data expression is substituted for another variable within a process expression. We cannot equationally define this operation completely when we are implementing the semantics, because the LOTOS syntax includes syntax for data expressions, which is user-definable. In the first implementation, this operation was defined at the metalevel, where a common, known syntax is used, the syntax for metarepresented terms. The fact of moving continuously between the object level and the metalevel when searching, that is, when proving a transition valid, allows the interleaving of the rewrite rules application at the object level (where rewrite rules represent the semantics), and the reduction of terms with substitutions at the metalevel (by means of equations defined there). Also a function to extract the variables in a process expression is used by the semantics, which presents the same problem and solution. In the second implementation we cannot use the same solution, because the search occurs completely at the object level, and sometimes the variables occurring in a process have to be known to check the condition of a rewrite. That is, we cannot go up to the metalevel, reduce the term, and go back to the object level to continue with the interrupted rule application. User-defined data expressions are included in the framework in both implementations by translating ACT ONE specifications into functional modules in Maude. It is in this translation where we solve the problem in the second implementation, by automatically adding equations that define at the object level the substitution and extraction of variables operations over new data expressions.

We have used Isabelle [12], a theorem prover and generic system for implementing logical formalisms, to represent the CCS semantics and the Hennessy-Milner modal logic, in order to compare this framework with ours. Isabelle has been designed as a logical framework and theorem prover, and therefore offers several automatic tools that help prove theorems, and which add a lot of power. In Maude, we only use the rewrite rules that define the semantics and the search that (blindly) uses them, being able to prove both sentences about CCS and the modal logic. Isabelle uses a higher-order logic to metarepresent the user logics. It is in this metalogic where resolution takes places. Due to the reflective property of rewriting logic, we can lower down this upper level, representing higher-order concepts in a first-order framework. In a sense, this comparison can be summarized by saying that we have shown in our CCS example how higher-order techniques can be used in a first-order framework by means of reflection; that is, reflection provides a first-order system like Maude with most of the power of a higher-order system like Isabelle.

As we said in the introduction, the techniques presented in this paper are quite general. Moreover, the second approach is a practical one, leading to reasonably efficient implementations. We have used it to implement a symbolic semantics for LOTOS [15]. It is also applicable to other kinds of operational semantics, from evaluation (big step) to computation (small step) semantics. We have successfully used it to implement the different semantics presented by Hennessy in [8].

Acknowledgements

We would like to thank Roberto Bruni for his comments on a previous version of this paper, Steven Eker for all his help and explanations in using the new version of Maude, and José Meseguer for encouraging and supporting all our research on executable semantic frameworks.

References

[1] R. Bruni. *Tile Logic for Synchronized Rewriting of Concurrent Systems*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999. Technical Report TD-1/99. `http://www.di.unipi.it/phd/tesi/tesi_1999/TD-1-99.ps.gz`.

[2] M. Calder and C. Shankland. A symbolic semantics and bisimulation for Full LOTOS. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proceedings of FORTE 2001, 21st International Conference on Formal Techniques for Networked and Distributed Systems*, pages 184–200. Kluwer Academic Publishers, 2001.

[3] G. Carabetta, P. Degano, and F. Gadducci. CCS semantics via proved transition systems and rewriting logic. In C. Kirchner and H. Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 253–272. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[4] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. Manual distributed as documentation of the Maude system, Computer Science Laboratory, SRI International. `http://maude.csl.sri.com/manual`, Jan. 1999.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 297–318. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[7] P. Degano, F. Gadducci, and C. Priami. A causal semantics for CCS via rewriting logic. Manuscript, Dipartimento di Informatica, Università di Pisa, submitted for publication, 2000.

[8] M. Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Willey & Sons, 1990.

[9] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, Aug. 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic, Second Edition, Volume 9*. Kluwer Academic Publishers, 2002. `http://maude.csl.sri.com/papers`.

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

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

[12] L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[13] M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic. In *Proc. of LFM'99: Workshop on Logical Frameworks and Meta-Languages*, Paris, France, Sept. 1999.

[14] C. Stirling. Modal and temporal logics for processes. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure vs Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 149–237. Springer-Verlag, 1996.

[15] A. Verdejo. Building tools for LOTOS symbolic semantics in Maude. In *Proceedings FORTE 2002*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[16] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In T. Bolognesi and D. Latella, editors, *Formal Methods For Distributed System Development. FORTE/PSTV 2000*, pages 351–366. Kluwer Academic Publishers, 2000.