IMPLEMENTING CCS IN MAUDE

Alberto Verdejo and Narciso Martí-Oliet
Depto. de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
{alberto,narciso}@sip.ucm.es

Abstract We explore the features of rewriting logic and the language Maude as a logical and semantic framework for representing both the semantics of CCS, and a modal logic for describing local capabilities of CCS processes. Although a rewriting logic representation of the CCS semantics was given previously, it cannot be directly executed in the default interpreter of Maude. Moreover, it cannot be used to answer questions such as which are the successors of a process after performing an action, which is used to define the semantics of the modal logic. Basically, the problems are the existence of new variables in the righthand side of the rewrite rules and the nondeterministic application of the semantic rules, inherent to CCS. We show how these problems can be solved by exploiting the reflective properties of rewriting logic, which allow controlling the rewriting process. This executable specification plus the reflective control of the rewriting process can be used to analyze CCS processes.

1. INTRODUCTION

Rewriting logic was introduced by Meseguer [8] as a unified model of concurrency in which several well-known models of concurrent systems can be represented in a common framework. This goal was further extended by Martí-Oliet and Meseguer [7] to the idea of rewriting logic as a *logical and semantic framework*. It was shown that many other logics, widely different in nature, can be represented inside rewriting logic in a natural and direct way. The general way in which such representations are achieved is by:

- Representing formulas or proof-theoretic structures such as sequents, as terms in an order-sorted equational data type whose equations express structural axioms natural to the logic in question.
- Representing the rules of deduction of a logic as rewrite rules that transform certain patterns of formulas into other patterns modulo the given structural axioms.

Similar techniques can be used to naturally specify and prototype many languages and systems in rewriting logic. In particular, the similarities between rewriting logic and structural operational semantics were noted by Meseguer [8] and further explored by Martí-Oliet and Meseguer [7]. As an illustrative example, Martí-Oliet and Meseguer [7] completely developed a representation of Milner's CCS [9] in rewriting logic. However, this representation cannot be directly executed in the default interpreter of Maude [3], a high-performance language and system supporting both equational and rewriting logic computation.

Basically, the problems are the existence of new variables in the right-hand side of the rewrite rules and the nondeterministic application of the semantic rules, inherent to CCS. We show how these problems can be solved by exploiting the reflective capabilities of rewriting logic and of Maude, that allow representing rewriting logic inside itself [2], and in particular controlling the rewriting process. This executable specification plus the reflective control of the rewriting process can be used to analyze CCS processes and to answer different questions such as which are the successors of a process after performing an action. In summary, we have managed to make the representation of CCS *executable* by using reflective techniques in such a way that it can be used to define in Maude the semantics of Hennessy-Milner modal logic [6].

In the rest of this section we review the representation of the CCS semantics in Maude, in order to see in detail how the problems we mentioned above arise.[1] First, we show the representation of the CCS syntax in two functional modules defining actions and processes.

```
(fmod ACTION is  protecting QID .
  sorts Label Act .  subsorts Qid < Label < Act .
  op tau : -> Act .  *** silent action
  op ~_ : Label -> Label .
  var N : Label .  eq ~ ~ N = N .
endfm)
(fmod PROCESS is  protecting ACTION .
  sorts ProcessId Process .  subsort Qid < ProcessId < Process .
  op 0 : -> Process .                              *** inaction
  op _._ : Act Process -> Process [prec 25] .      *** prefix
  op _+_ : Process Process -> Process [prec 35] .  *** summation
  op _|_ : Process Process -> Process [prec 30] .  *** composition
  op _`[_/_`] : Process Label Label -> Process [prec 20] .
                       *** relabelling: [b/a] relabels "a" to "b"
  op _\_ : Process Label -> Process [prec 20] .    *** restriction
endfm)
```

[1] For lack of space, the reader is assumed to be familiar with the Maude syntax [3], as well as CCS and its operational semantics [9].

Full CCS is represented, including (possibly recursive) process definitions by means of *contexts*.

```
(fmod CCS-CONTEXT is  protecting PROCESS .
  sorts BadProcess Context BadContext .
  subsort Process < BadProcess .  subsort Context < BadContext .
  op _=def_ : ProcessId Process -> Context .
  op nil : -> Context .
  op _&_ : BadContext BadContext -> BadContext [assoc comm id: nil] .
  op _definedIn_ : ProcessId Context -> Bool .
  op def : ProcessId Context -> BadProcess .
  op not-defined : -> BadProcess .
  op context : -> Context .
  vars X X' : ProcessId .  var P : Process .  var C : Context .
  cmb (X =def P) & C : Context if not(X definedIn C) .
  eq X definedIn nil = false .
  eq X definedIn ((X' =def P) & C) = (X == X') or (X definedIn C) .
  eq def(X, nil) = not-defined .
  eq def(X, ((X' =def P) & C)) = if X == X' then P else def(X, C) fi .
endfm)
```

The general idea for implementing in rewriting logic the operational semantics of CCS, is to translate each semantic rule into a rewrite rule where either the premises are rewritten to the conclusion, or the conclusion is rewritten to the premises. In previous work [7] the first approach was followed; here we adopt the second one, because we want to be able to prove in a bottom-up way that a given transition is valid in CCS.

The CCS transition $P \xrightarrow{a} P'$ is represented in Maude by the term P `--` a `->` P', of sort `Judgement`, built by means of the operator

```
  op _--_->_ : Process Act Process -> Judgement [prec 50] .
```

In general, a semantic rule has a conclusion and a set of premises, each one represented by a judgement. So we need a sort to represent sets of judgements:

```
  sort JudgementSet .  subsort Judgement < JudgementSet .
  op emptyJS : -> JudgementSet .
  op __ : JudgementSet JudgementSet -> JudgementSet
          [assoc comm id: emptyJS prec 60] .
  var J : Judgement .  eq J J = J .
```

The union constructor is written with empty syntax (`__`), and declared associative (`assoc`), commutative (`comm`), and with identity element the empty set (`id: emptyJS`). Matching and rewriting take place *modulo* such properties. Idempotency is specified by means of an explicit equation.

A semantic rule is implemented as a rewrite rule where the singleton set consisting of the judgement representing the conclusion is rewritten to the set consisting of the judgements representing the premises.

For example, for the restriction operator of CCS, we have the rule[2]

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

and for the axiom schema defining the prefix operator we have

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

Thus, a transition $P \xrightarrow{a} P'$ is possible in CCS if and only if the judgement representing it can be rewritten to the empty set of judgements by rewrite rules of the form described above that define the operational semantics of CCS in a backwards search fashion.

However, we have found problems while working with this approach in the current version of Maude. The first one is that sometimes new variables appear in the premises which are not in the conclusion. For example, in one of the semantic rules for the parallel operator we have

```
rl [par] :     P | Q -- tau -> P' | Q'
        => ---------------------------------
            P -- L -> P'   Q -- ~ L -> Q' .
```

where `L` is a new variable in the righthand side of the rewrite rule. 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.

Another problem is that sometimes several rules can be applied to rewrite a judgement. For example, for the summation operator we have, because of its intrinsic nondeterminism,[3]

```
rl [sum] :   P + Q -- A -> P'      rl [sum] :   P + Q -- A -> Q'
        => ------------------              => --------------------
               P -- A -> P' .                      Q -- A -> Q' .
```

In general, not all of these possibilities lead to an empty set of judgements. So we have to deal with the whole tree of possible rewritings of a judgement, searching if one of the branches leads to `emptyJS`.

In Section 2, we show how these problems can be solved in the current version of the Maude system by using reflection, obtaining an executable semantics, where we can prove if a transition $P \xrightarrow{a} P'$ is possible.

---

[2] Using the fact that text beginning with `---` is a comment in Maude, rules are displayed in such a way as to emphasize the correspondence with the usual presentation in textbooks, although in this case the conclusion is above the horizontal line.

[3] Only one rule is enough by declaring `_+_` to be commutative. In this case, nondeterminism appears because of possible multiple matches (modulo commutativity) against pattern `P + Q`.

In Section 3, we extend this representation in order to be able to answer different kinds of questions, such as if process P can perform action a (and we do not care about the process it becomes), or which are the *successors* of a process P after performing actions in a given set As, that is,

$$succ(P, As) = \{P' \mid P \xrightarrow{a} P' \wedge a \in As\}.$$

In Section 4 we show how we can define in Maude the semantics of the Hennessy-Milner modal logic for describing local capabilities of CCS processes.

2. EXECUTABLE CCS SEMANTICS

In this section we show how the problem of new variables in the right-hand side of a rewrite rule is solved by using the concept of *explicit metavariables* presented by Stehr and Meseguer [11], and how nondeterministic rewriting is controlled by using a *search strategy* [1, 4].

New variables in the righthand side of a rule represent "unknown" values when we are rewriting; by using metavariables we make explicit this lack of knowledge. The semantics with explicit metavariables has to bind them to concrete values when these values become known.

For the moment, metavariables are only needed as actions in the judgements, so we declare a new sort for metavariables as actions:

```
sort MetaVarAct .
op ?‘(_‘)A : Qid -> MetaVarAct .
var NEW1 : Qid .
```

We also introduce a new sort `Act?` of "possible actions," and modify the operator for building judgements in order to deal with it:

```
sort Act? .  subsorts Act MetaVarAct < Act? .
var ?A : MetaVarAct .  var A? : Act? .
op _--_->_ : Process Act? Process -> Judgement [prec 50] .
```

As mentioned above, a metavariable will be bound when its concrete value becomes known, so we need a new judgement stating that a metavariable is bound to a concrete value

```
op ‘[_:=_‘] : MetaVarAct Act -> Judgement .
```

and a way to propagate this binding to the rest of judgements where the bound metavariable may be present. Since this propagation has to reach all the judgements in the current state of the inference process, we introduce an operation to enclose the set of judgements, and a rule to propagate a binding

```
op ‘{‘{_‘}‘} : JudgementSet -> Configuration .
var JS : JudgementSet .
```

```
rl [bind] : {{ [?A := A] JS }} => {{ <act ?A := A > JS }} .
```

where we use several overloaded, auxiliary functions `<act_:=_>_` to perform the substitutions (see complete specification in the full version [13]).

Now we are able to redefine the rewrite rules implementing the CCS semantics, taking care of metavariables. For the prefix operator we maintain the previous axiom schema and add a new rule for the case when a metavariable appears in the judgement

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

Note how the metavariable `?A` present in the lefthand side judgement is bound to the concrete action `A` taken from the process `A.P`. This binding will be propagated to any other judgement in the set of judgements containing `A.P -- ?A -> P`.

For the summation operator, we generalize the rules allowing a more general variable `A?` of sort `Act?`, since the behavior is the same independently of whether a metavariable or an action appears in the judgement:

```
rl [sum] :  P + Q -- A? -> P'        rl [sum] :  P + Q -- A? -> Q'
       => -------------------               => -------------------
             P -- A? -> P' .                      Q -- A? -> Q' .
```

Nondeterminism is again present; we deal with it in Section 2.1.

For the parallel operator, there are two rules for the cases when one of the composed processes performs an action on its own,

```
rl [par] : P | Q -- A? -> P' | Q     rl [par] : P | Q -- A? -> P | Q'
       => ----------------------            => ------------------------
             P -- A? -> P' .                      Q -- A? -> Q' .
```

and two additional rules dealing with the case when communication happens between both processes,

```
rl [par] :            P | Q -- tau -> P' | Q'
       => ---------------------------------------------
           P -- ?(NEW1)A -> P'   Q -- ~ ?(NEW1)A -> Q' .

rl [par] :                P | Q -- ?A -> P' | Q'
       => ------------------------------------------------------------
           P -- ?(NEW1)A -> P'   Q -- ~ ?(NEW1)A -> Q'  [?A := tau] .
```

where we have overloaded the `~` operator `op ~_ : Act? -> Act? .`

Note how the term `?(NEW1)A` is used to represent a *new metavariable*. Rewriting has to be controlled by a *strategy* that instantiates the variable `NEW1` with a new (quoted) identifier each time one of the above rules is applied, in order to build *new* metavariables. The strategy presented in Section 2.1 does this as well as implementing the search in the tree of possible rewritings.

There are two rules dealing with the restriction operator of CCS, depending on whether an action or a metavariable occurs in the lefthand side judgement

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

rl [res] :         P \ L -- ?A -> P' \ L
       => ----------------------------------------
           P -- ?A -> P'  [?A =/= L]  [?A =/= ~ L] .
```

In the latter case, we cannot use a conditional rewrite rule as in the former case, because the condition `(?A =/= L) and (?A =/= ~ L)` cannot be checked until we know the concrete value of the metavariable `?A`. Hence, we have to add a new kind of judgement used to state constraints between metavariables, which is eliminated when it is fulfilled,

```
op `[_=/=_`] : Act? Act? -> Judgement .
crl [dist] : [A =/= A'] => emptyJS  if A =/= A' .
```

where (normal) actions are used.

For the relabelling operator of CCS we have similar rewrite rules, and for process identifiers we only need the generalization of the original rule by means of a more general variable `A?`.

```
crl [def] :         X -- A? -> P'?
         => ------------------------------
             def(X, context) -- A? -> P'?   if (X definedIn context) .
```

Using these rules, we can begin to pose some questions about the capability of a process to perform an action. For example, we can ask if the process `'a.'b.0` can perform action `'a` (becoming `'b.0`) by rewriting the configuration composed of a judgement representing that transition:

```
Maude> (rew {{ 'a . 'b . 0 -- 'a -> 'b . 0 }} .)
Result Configuration : {{ emptyJS }}
```

Since a configuration consisting of the empty set of judgements is reached, we can conclude that the transition is possible.

However, if we ask if the process `'a.0 + 'b.0` can perform action `'b` becoming process `0`, we get as result

```
Maude> (rew {{ 'a . 0 + 'b . 0 -- 'b -> 0 }} .)
Result Configuration : {{ 'a . 0 -- 'b -> 0 }}
```

representing that the given transition is not possible, which is not the case. The problem is that `{{'a.0 + 'b.0 -- 'b -> 0}}` can be rewritten in two different ways, and only one of them leads to a configuration consisting of the empty set of judgements.

Therefore, we need a strategy to search the tree of all possible rewrites.

2.1. SEARCHING IN THE TREE OF REWRITINGS

Rewriting logic is reflective [2], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$, and we then have the following equivalence: $\mathcal{R} \vdash t \longrightarrow t' \;\Leftrightarrow\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$.

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`, where Maude terms are reified as elements of a data type `Term`, Maude modules are reified as terms in a data type `Module`, the process of reducing a term to normal form is reified by a function `meta-reduce`, and the process of applying a rule of a system module to a subject term is reified by a function `meta-apply` [3].

In this section we show how the reflective properties of Maude [2] can be used to control the rewriting of a term and the search in the tree of possible rewritings of a term. The depth-first strategy is based on previous work [1, 4], although modified to deal with the substitution of metavariables explained in the previous section.

The module implementing the search strategy is parameterized with respect to a constant equal to the metarepresentation of the Maude module which we want to work with. Hence, we define a parameter theory with a constant `MOD` representing the module, and a constant `labels` representing the list of labels of rewrite rules to be applied:

```
(fth AMODULE is  including META-LEVEL .
  op MOD : -> Module .
  op labels : -> QidList .
endfth)
```

The module containing the strategy, extending `META-LEVEL`, is then the parameterized module `SEARCH[M :: AMODULE]`.

Since we are defining a strategy to search a tree of possible rewritings, we need a notion of search goal. For the strategy to be general enough, we assume that the module `MOD` has an operation `ok` (defined at the object level), which returns a value of sort `Answer` such that

- `ok(T) = solution` means that the term `T` is one of the terms we are looking for, that is, `T` denotes a solution;
- `ok(T) = no-solution` means that the term `T` is not a solution and no solution can be found below `T` in the search tree;
- `ok(T) = maybe-sol` means that `T` is not a solution, but we do not know if there are solutions below it.

The strategy controls the possible rewritings of a term by means of `meta-apply`. `meta-apply(M,T,L,S,N)` applies (discarding the first `N` successful matches) a rule of module `M` with label `L`, partially instantiated with substitution `S`, to the term `T`. It returns the resulting fully reduced term and the representation of the match used in the reduction.

We saw before the necessity of instantiating the new variables in the righthand side of a rewrite rule in order to create new metavariables. We have to provide a substitution in such a way that the rules are always applied without new variables in the righthand side. For simplicity we will assume that a rule has at most three new variables called `NEW1`, `NEW2`, and `NEW3`. These variables are then substituted by new identifiers, which are quoted numbers. Hence, we define a new operation `meta-apply'` which receives the greatest number used to substitute variables in `T` and uses three new numbers to create three new (metarepresented) identifiers.

```
vars N M : MachineInt .  var L : Qid .  var T : Term .
op subst : MachineInt -> Substitution .
eq subst(M) = (('NEW1@Qid <- {conc('' , index(' , M + 1))}'Qid);
               ('NEW2@Qid <- {conc('' , index(' , M + 2))}'Qid);
               ('NEW3@Qid <- {conc('' , index(' , M + 3))}'Qid)) .
op meta-apply' : Term Qid MachineInt MachineInt -> Term .
eq meta-apply'(T,L,N,M) = extTerm(meta-apply(MOD,T,L,subst(M),N)) .
```

`meta-apply'` returns *one* of the possible one-step rewritings at the top level of a given term. Our first step is to define an operation `allRew` that returns *all* the possible *one-step sequential* rewritings [8] of a given term by using rewrite rules with labels in the list `labels`. The third argument of `allRew` represents the greatest number `M` used to substitute new variables in `T`. There is a `TermList` sort in module `META-LEVEL`, but it does not have an identity element, which we need to represent the case when no rule can be applied. So we extend it as follows:

```
op ~ : -> TermList .  var TL : TermList .
eq ~, TL = TL .  eq TL, ~ = TL .
```

The operations needed to find all the possible rewritings, and their definitions, are as follows:

```
op allRew : Term QidList MachineInt  -> TermList .
op topRew : Term Qid MachineInt MachineInt -> TermList .
op lowerRew : Term Qid MachineInt -> TermList .
op rewArguments : Qid TermList TermList Qid MachineInt -> TermList .
op rebuild : Qid TermList TermList TermList -> TermList .
var LS : QidList . vars C S OP : Qid . vars Before After : TermList .
eq allRew(T, nil, M) = ~ .
eq allRew(T, L LS, M) = topRew(T, L, 0, M), *** rew at the top of T
                        lowerRew(T, L, M),  *** rew of subterms
                        allRew(T, LS, M) .  *** rew with labels LS
```

```
eq topRew(T, L, N, M) =
    if meta-apply'(T, L, N, M) == error* then ~
    else (meta-apply'(T, L, N, M) , topRew(T, L, N + 1, M)) fi .
eq lowerRew({C}S, L, M) = ~ .
eq lowerRew(OP[TL], L, M) = rewArguments(OP, ~, TL, L, M) .
eq rewArguments(OP, Before, T, L, M) =
    rebuild(OP, Before, allRew(T, L, M), ~) .
eq rewArguments(OP, Before, (T, After), L, M) =
    rebuild(OP, Before, allRew(T, L, M), After) ,
    rewArguments(OP, (Before, T), After, L, M) .
eq rebuild(OP,Before,~,After) = ~ .
eq rebuild(OP,Before,T,After) = meta-reduce(MOD,OP[Before,T,After]) .
eq rebuild(OP,Before,(T, TL),After) =
   meta-reduce(MOD, OP[Before,T,After]), rebuild(OP,Before,TL,After) .
```

Now we can define a strategy to search in the (conceptual) tree of all possible rewritings of a term `T` for a term that satisfies the `ok` predicate. Each node of the search tree is a pair, whose first component is a term and whose second component is a number representing the greatest number used as identifier for new variables in the process of rewriting the term. The tree nodes that have been generated but not yet been checked are maintained in a sequence.

```
sorts Pair PairSeq .  subsort Pair < PairSeq .
op <_‘,_> : Term MachineInt -> Pair .
op nil : -> PairSeq .
op _|_ : PairSeq PairSeq -> PairSeq [assoc id: nil] .
var PS : PairSeq .
```

We need an operation to build these pairs from the list of terms produced by `allRew`:

```
op buildPairs : TermList MachineInt -> PairSeq .
eq buildPairs(~, N) = nil .  eq buildPairs(T, N) = < T , N > .
eq buildPairs((T, TL), N) = < T , N > | buildPairs(TL, N) .
```

The operation `rewDepth` starts the search by calling the operation `rewDepth'` with the root of the search tree. `rewDepth'` returns the first solution found in a depth-first way. If there is no solution, the `error*` term is returned.

```
op rewDepth : Term -> Term .
op rewDepth' : PairSeq -> Term .
eq rewDepth(T) = rewDepth'(< meta-reduce(MOD, T), 0 >) .
eq rewDepth'(nil) = error* .
eq rewDepth'(< T , N > | PS) =
   if meta-reduce(MOD, 'ok[T]) == {'solution}'Answer then T
   else (if meta-reduce(MOD, 'ok[T]) == {'no-solution}'Answer then
              rewDepth'(PS)
         else rewDepth'(buildPairs(allRew(T,labels,N),N + 3) | PS)
         fi) fi .
```

Now we can test the CCS semantics with some examples using different judgements. First, we define a module `CCS-OK` extending the CCS syntax and semantic rules by defining some process constants to be used in the examples, and the predicate `ok` that states when a configuration is a solution. In this case a configuration denotes a solution when it is the empty set of judgements, representing that the set of judgements at the beginning is provable by means of the semantic rules.

```
(mod CCS-OK is  including CCS-SEMANTICS .
  ops p1 p2 : -> Process .
  eq p1 = ('a . 0) + ('b . 0 | ('c . 0 + 'd . 0)) .
  eq p2 = ('a . 'b . 0 | (~ 'c . 0) ['a / 'c]) \ 'a .
  sort Answer .
  ops solution  no-solution  maybe-sol : -> Answer .
  op ok : Configuration -> Answer .
  var JS : JudgementSet .
  eq ok({{ emptyJS }}) = solution .
  ceq ok({{ JS }}) = maybe-sol if JS =/= emptyJS .
endm)
```

In order to instantiate the parameterized generic module `SEARCH`, we use the Full Maude `up` function to obtain the metarepresentation of module `CCS-OK`, and then we declare a view [3]:

```
(mod META-CCS is  including META-LEVEL .
  op METACCS : -> Module .
  eq METACCS = up(CCS-OK) .
endm)
(view ModuleCCS from AMODULE to META-CCS is
  op MOD to METACCS .
  op labels to ('bind 'pref 'sum 'par 'res 'dist 'rel 'def) .
endv)
(mod SEARCH-CCS is
  protecting SEARCH[ModuleCCS] .
endm)
```

Now we can test the examples. First we can prove that process `p1` can perform action `'c` becoming `'b.0 | 0`.[4]

```
Maude> (red rewDepth(‾{{ p1 -- 'c -> 'b . 0 | 0 }}‾)). )
Result Term : ‾{{ emptyJS }}‾
```

We can also prove that `p2` cannot perform action `'a` (but see later).

```
Maude> (red rewDepth(‾{{p2 -- 'a -> ('b.0 | (~'c.0) ['a/'c])\'a}}‾)).)
Result Term : error*
```

[4] We refer to the Maude manual [3] for indications about how to introduce metarepresented terms. Here we use $\overline{t}$ for the metarepresentation of term t.

In these examples, we have had to provide the resulting process. In the positive proof there is no problem, but in the negative proof, that is, that `p2` cannot perform action `'a`, the given proof is not completely correct: We have proved that process `p2` cannot perform action `'a` *becoming* `('b.0 | (~'c.0) ['a/'c])\'a`, but we have not proved that there is no way in which `p2` can execute action `'a`. We will see in the next section how this can be proved.

3. HOW TO OBTAIN NEW RESULTS

We are now interested in answering questions such as: Can process P perform action a (without caring about the process it becomes)? That is, we want to know if $P \xrightarrow{a} P'$ is possible, but P' is *unknown*. This is the same problem we found when new variables appear in the premises of a semantic rule. The solution, as we did with actions, is to define metavariables as processes, adding a new sort of possible processes, and modifying the operator used to build the basic transition judgements.

```
sort MetaVarProc .  op ?`(_`)P : Qid -> MetaVarProc .
sort Process? .  subsorts Process MetaVarProc < Process? .
var ?P : MetaVarProc .
op _--_->_ : Process? Act? Process? -> Judgement [prec 50] .
```

We also have to define a new kind of judgements that binds metavariables with processes, a rule to propagate these bindings, and operations that perform the substitution (see full version [13]). For the CCS operators, new rules have to be added to deal with metavariables in the second process of the transition judgement. For example, for the prefix operator we have to add two new rules:

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

Now, we can prove that process `p2` *cannot* perform action `'a`, by rewriting the judgement `p2 -- 'a -> ?('any)P`, where the metavariable `?('any)P` means that we do not care about the resulting process.

```
Maude> (red rewDepth({{ p2 -- 'a -> ?('any)P }}) .)
Result Term : error*
```

Another interesting question is which are the *successors* of a process P after performing actions in a given set As, that is,

$$succ(P, As) = \{P' \mid P \xrightarrow{a} P' \wedge a \in As\}.$$

Since we can use metavariables as processes, we have to rewrite a judgement like `P -- A -> ?('proc)P` instantiating the variable `A` with actions in the given set. Those rewritings will bind the metavariable

`?('proc)P` with the successors of `P`, but we find two problems. The first one is that we lose the bindings between metavariables and processes when they are substituted by applying the rewrite rule `bind`. To solve this, we have to modify the operator to build configurations, by also keeping a set of bindings already produced, which will be saved by the `bind` rule:

```
op `{`{_|_`}`} : JudgementSet JudgementSet -> Configuration  .
rl [bind] : {{ [?P := P] JS | JS' }} =>
            {{ (<proc ?P := P > JS) | [?P := P] JS' }} .
```

We have to change also the function `ok`, adding this new argument.

Another problem is that `rewDepth` only returns one solution, but we can modify it in order to get all the solutions, that is, in order to explore the whole tree of rewritings finding *all* the nodes that satisfy the function `ok`. The operation `allSol` returns a set with all the solutions.

```
sort TermSet .  subsort Term < TermSet .
op `{`} : -> TermSet .
op _U_ : TermSet TermSet -> TermSet [assoc comm id: {}] .
ceq T U T' = T if meta-reduce(MOD, '_==_[T, T']) == {'true}'Bool .
op allSol : Term -> TermSet .
eq allSol(T) = allSolDepth(< meta-reduce(MOD,T), 0 >) .
op allSolDepth : PairSeq -> TermSet .
eq allSolDepth(nil) = {} .
eq allSolDepth(< T , N > | PS) =
  if meta-reduce(MOD, 'ok[T]) == {'solution}'Answer then
              (T U allSolDepth(PS))
  else (if meta-reduce(MOD, 'ok[T]) == {'no-solution}'Answer then
            allSolDepth(PS)
       else allSolDepth(buildPairs(allRew(T,labels,N),N + 3) | PS)
       fi)
  fi .
```

Now we can define (in an extension of module `SEARCH-CCS`) a function `succ` which, given the metarepresentation of a process and a set of metarepresentations of actions, returns the set of metarepresentations of the process successors.

```
op succ : Term TermSet -> TermSet .
eq succ(T, {}) = {} .
eq succ(T, A U AS) = filter(allSol(
        ''{`{_|_`}`} [ '_--_->_ [ T, A, '?`(_`)P [{''proc}'Qid]] ,
        {'emptyJS}'JudgementSet ]), '?`(_`)P [{''proc}'Qid])
     U succ(T,AS) .
```

where `filter` (see full version [13]) is used to remove all the bindings involving metavariables different from `?('proc)P`.

4. MODAL LOGIC FOR CCS PROCESSES

In this section we show how we can define in Maude the semantics of a modal logic for CCS processes by using the functions of previous sections. We introduce a *modal* logic for describing local capabilities of CCS processes. This logic, which is a version of Hennessy-Milner logic [6], and its semantics are presented by Stirling [12]. Formulas are as follows:

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

where K is a set of actions. The satisfaction relation describing when a process P satisfies a property Φ, $P \models \Phi$, is defined as follows:

$$\begin{array}{ll} P \models \mathtt{tt} & \\ P \models \Phi_1 \wedge \Phi_2 & \text{iff } P \models \Phi_1 \text{ and } P \models \Phi_2 \\ P \models \Phi_1 \vee \Phi_2 & \text{iff } P \models \Phi_1 \text{ or } P \models \Phi_2 \\ P \models [K]\Phi & \text{iff } \forall\, Q \in \{P' \mid P \xrightarrow{a} P' \wedge a \in K\}\,.\, Q \models \Phi \\ P \models \langle K \rangle \Phi & \text{iff } \exists\, Q \in \{P' \mid P \xrightarrow{a} P' \wedge a \in K\}\,.\, Q \models \Phi \end{array}$$

First, we define a sort `HMFormula` of modal logic formulas and operations to build these formulas:

```
(mod MODAL-LOGIC is  protecting CCS-SUCC .
  sort HMFormula .
  ops tt ff : -> HMFormula .
  ops _/\_  _\/_ : HMFormula HMFormula -> HMFormula .
  ops `[_`]_  <_>_ : TermSet HMFormula -> HMFormula .
```

We define the modal logic semantics in the same way as we did with the CCS semantics, that is, by defining rewrite rules that rewrite a judgement $P \models \Phi$ into the set of judgements which have to be fulfilled:

```
  op _|=_ : Term HMFormula -> Judgement .
  op forall : TermSet HMFormula -> JudgementSet .
  op exists : TermSet HMFormula -> JudgementSet .
  var P : Term .  vars K PS : TermSet .  vars Phi Psi : HMFormula .
  rl [true] : P |= tt          =>  emptyJS .
  rl [and]  : P |= Phi /\ Psi  =>  (P |= Phi) (P |= Psi) .
  rl [or]   : P |= Phi \/ Psi  =>  P |= Phi .
  rl [or]   : P |= Phi \/ Psi  =>  P |= Psi .
  rl [box]  : P |= [ K ] Phi   =>  forall(succ(P, K), Phi) .
  rl [diam] : P |= < K > Phi   =>  exists(succ(P, K), Phi) .
  eq forall({}, Phi)     = emptyJS .
  eq forall(P U PS, Phi) = (P |= Phi) forall(PS, Phi) .
  rl [ex] : exists(P U PS, Phi) => P |= Phi .
```

These rules are also nondeterministic. For example, the application of the two rules `or` is nondeterministic because they have the same left-hand side, and the rule `ex` is also nondeterministic because of multiple

matchings modulo associativity and commutativity. We can instantiate the module `SEARCH` in Section 2.1 with the metarepresentation of the module containing the definition of the modal logic semantics.

As an example, we show some modal formulas satisfied by a vending machine `'Ven` [12] defined in a CCS context as

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

and how they can be proved in Maude.

One of the properties that the vending machine fulfills is that a button cannot be depressed initially, that is, before money is deposited.

```
Maude> (red rewDepth('Ven |= [ 'big U 'little ] ff) .)
Result Term : emptyJS
```

Another interesting property of `'Ven` is that after a `'2p` coin is inserted, the little button cannot be depressed whereas the big one can:

```
Maude> (red rewDepth('Ven |= ['2p](([']little] ff) /\ (<'big> tt))) .)
Result Term : emptyJS
```

5. CONCLUSION

We have represented the CCS structural operational semantics in rewriting logic in a general way, solving the problems of new variables in the righthand side of the rules and nondeterminism by means of the reflective features of rewriting logic and its realization in the Maude module `META-LEVEL`.

We have seen how the semantics can be extended to answer questions about the capability of a process to perform an action, by considering metavariables as processes in the same way as we had done with actions. Having metavariables as processes allows also answering which are the successors of a process after performing an action, and this allows a representation of the semantics of the Hennessy-Milner modal logic in a similar way to its mathematical definition.

In the full version of this paper [13], in addition to more details, we have extended (using the same techniques) the semantics representation to sequences of actions or *traces*. We have also implemented *weak transition semantics*, $P \stackrel{a}{\Longrightarrow} P'$, which does not observe τ transitions. Finally, we have extended the modal logic representation introducing new modalities $[\![K]\!]$ and $\langle\langle K \rangle\rangle$ defined by means of the weak transition relation.

Other specific-purpose tools, such as the Concurrency Workbench [5], are more efficient and expressive than our tool, which is much in the

style of a prototype, where we can play not only with processes and their capabilities, but with the semantics, represented at a very high mathematical level, adding or modifying some rules.

Although our implementation is much in the style of logic programming, one advantage is the possibility of working with algebraic specifications modulo equational axioms. Moreover, other strategies could be employed rather than depth-first, still keeping the same underlying specification.

Work in progress applies all these techniques to the specification language E-LOTOS, developed within ISO for the formal specification of open distributed concurrent systems [10].

Acknowledgements We are grateful to J. Meseguer and the referees for their helpful comments on earlier versions of this paper.

References

[1] R. Bruni. *Tile Logic for Synchronized Rewriting of Concurrent Systems.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999.

[2] M. Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language.* PhD thesis, University of Navarre, 1998.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic.* SRI International, Jan. 1999, revised Aug. 1999.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Using Maude. In *Proc. FASE 2000*, LNCS 1783. Springer, 2000.

[5] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, Jan. 1993.

[6] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, Jan. 1985. .

[7] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, 1993. To appear in *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

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

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

[10] J. Quemada, editor. Final committee draft on Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG7 Project 1.21.20.2.3., May 1998.

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

[12] C. Stirling. Modal and temporal logics for processes. In *Logics for Concurrency: Structure vs Automata*, LNCS 1043, pages 149–237. Springer, 1996.

[13] A. Verdejo and N. Martí-Oliet. Executing and verifying CCS in Maude. Technical Report 99-00, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Feb. 2000.