Executing and Verifying CCS in Maude*

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

21st November 2001

Abstract

We explore the features of rewriting logic and, in particular, of the rewriting logic 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 current 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 Hennessy-Milner 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. We also show how the semantics can be extended to traces of actions and to the CCS weak transition relation. This executable specification plus the reflective control of the rewriting process can be used to analyze CCS processes.

Keywords
Rewriting logic, Maude, semantic framework, internal strategies, CCS, Hennessy-Milner modal logic.

1 Introduction

Rewriting logic [18] was introduced in [17] 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 in [16] 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:

*Research supported by CICYT project *Desarrollo Formal de Sistemas Distribuidos* (TIC97–0669–C03–01).

- Representing formulas or, more generally, 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 [22] were noted in [17] and further explored in [16]. As an illustrative example, the paper [16] completely develops a representation of Milner's CCS [20] in rewriting logic, extending ideas first introduced in [19]. However, this representation of CCS cannot be directly executed in the current default interpreter of Maude [8, 6], a high-level language and high-performance system supporting both equational and rewriting logic computation. Maude should be viewed as a *metalanguage* in which the syntax and semantics of other languages, including formal specification languages, can be formally defined [5, 9].

Basically, the problems with the CCS specification 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 two problems can be solved by exploiting the reflective capabilities of rewriting logic and of Maude, that allow representing rewriting logic inside itself [10, 4], and in particular controlling the rewriting process. Moreover, we show how the semantics can be extended to traces of actions and to the CCS weak transition relation [20]. 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 [14].

In the rest of this introduction we review the representation of the CCS semantics in Maude, in order to see in detail how the two problems we mentioned above arise. At the same time, we introduce Maude syntax. We use the Full Maude extension [13] to take advantage of parameterization mechanisms.[1] To begin with, we show the representation of the CCS syntax in two *functional modules*, that is, equational theories used to specify algebraic data types, defining actions and processes. We declare `sort(s)`, `subsort(s)`, and operators `op(s)`, which have user-definable syntax.[2] The operators' precedence is set by means of the attribute `prec`. Equations are declared with the keywords `eq` or `ceq` (for conditional ones). The imported module `QID` is a useful built-in module providing quoted identifiers that in this case represent CCS labels and process identifiers.

```
(fmod ACTION is
  protecting QID .
  sorts Label Act .
  subsorts Qid < Label < Act .
  op tau : -> Act .    *** silent action
```

[1] The Maude system, including the Full Maude extension, and its documentation are available on the Maude web site at `http://maude.csl.sri.com`.

[2] Due to parsing restrictions, some characters (`[ ] { } ( ) ,`) have to be preceded by character ` when declaring them in Full Maude.

```
  op ~_ : Label -> Label .
  var N : Label .
  eq ~ ~ N = N .
endfm)

(fmod PROCESS is
  protecting ACTION .
  sorts ProcessId Process .
  subsorts 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)
```

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. We use a conditional membership axiom (`cmb`) to establish which terms of sort `BadContext` 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, it returns the bad process `not-defined`.

```
(fmod CCS-CONTEXT is
  protecting PROCESS .
  sorts BadProcess Context BadContext .
  subsort Process < BadProcess .
  subsort Context < BadContext .
  op _=def_ : ProcessId Process -> Context [prec 40] .
  op nil : -> Context .
  op _&_ : BadContext BadContext -> BadContext
             [assoc comm id: nil prec 42] .
  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)) = P .
  ceq def(X, ((X' =def P) & C)) = def(X, C) if X =/= X' .
endfm)
```

The constant `context` is used to keep the definitions of the process identifiers used in each CCS specification. It is declared here because it is needed in the semantics (Section 2.1), but it will be defined in an example below (see Section 2.4).

The general idea, as stated in [16], for implementing in rewriting logic the operational semantics of CCS, is to translate each semantic rule either into a rewrite rule where the premises are rewritten to the conclusion, or into a rewrite rule where the conclusion is rewritten to the premises. In [16] the first approach was followed. In this paper we adopt the second one, because we want to be able to prove in a bottom-up goal-directed way that a given transition is valid in CCS. There is another possibility, that is not considered in this paper, where the transition $P \stackrel{a}{\longrightarrow} P'$ is interpreted as a rewrite, so that the operational semantic rules become conditional rewrite rules where the premises form the condition [16].

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

```
sort Judgement .
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] .
```

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

```
var J : Judgement .
eq J J = J .
```

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. Rewrite rules (introduced with the keywords `rl` or `crl`) are declared in *system modules*, which are rewrite theories specifying concurrent systems. Hence, system modules define the *dynamic* aspects of systems, whereas functional modules define *static* data types.

For example, for the restriction operator of CCS, we have the semantic rule,

$$\frac{P \stackrel{a}{\longrightarrow} P'}{P \setminus l \stackrel{a}{\longrightarrow} P' \setminus l}\ [\, a \neq l \wedge a \neq \bar{l}\,]$$

which is translated to the following rewrite rule where, using the fact that text beginning with `---` is a comment in Maude, the rule is 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):

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

As another example, the axiom schema

$$\frac{}{a.P \xrightarrow{a} P}$$

defining the behavior of the prefix operator gives rise to the rewrite rule

```
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 two problems while working with this approach in the current version of the Maude system. The first one is that sometimes new variables appear in the premises which are not present 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 current 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,

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

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

Only one rule is enough by declaring the operator `_+_` to be commutative. However, in the commutative case, nondeterminism appears because of possible multiple matches (modulo commutativity) against the pattern `P + 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.

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,

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

In Sections 4 and 5 we extend the CCS semantics to sequences of actions (or *traces*) and to the weak transition semantics, which does not observe τ transitions.

In Section 6 we show how we can define in Maude the semantics of the Hennessy-Milner modal logic for describing local capabilities of CCS processes.

In Section 7 we present how the CCS semantics and the Hennessy-Milner modal logic can be represented and reasoned about within the theorem prover Isabelle [21], and compare this framework with ours.

We show in Section 8 how the ideas presented in this paper are being applied to build a tool for working with Full LOTOS specifications, based on a symbolic semantics [3]. Although this section presents some improvements, we have not applied them to the CCS case in this paper because we want to present step by step the process of defining an executable operational semantics in Maude, from the basis to the complex results.

In Section 9 we draw some conclusions.

2 Executable CCS semantics in Maude

In this section we show how the problem of new variables in the righthand side of a rewrite rule is solved by using the concept of *explicit metavariables* presented in [23], and how nondeterministic rewriting is controlled by means of a *search strategy* [2, 7].

2.1 Definition of the executable semantics

New variables in the righthand side of a rewrite 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 time being, 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,"

```
sort Act? .
subsorts Act MetaVarAct < Act? .
var ?A : MetaVarAct .
var A? : Act? .
```

and modify the operator for building judgements in order to deal with this new sort of actions:

```
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

```
sort Configuration .
op `{`{_`}`} : JudgementSet -> Configuration .
var JS : JudgementSet .
rl [bind] : {{ [?A := A] JS }} => {{ <act ?A := A > JS }} .
```

where we use the following overloaded auxiliary functions in order to perform the corresponding substitutions (see Appendix A)

```
op <act_:=_>_ : MetaVarAct Act Act? -> Act? .
op <act_:=_>_ : MetaVarAct Act Judgement -> Judgement .
op <act_:=_>_ : MetaVarAct Act JudgementSet -> JudgementSet .
```

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

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

and add a new rule for the case when a metavariable appears in the judgement

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

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

Nondeterminism is again present; we will deal with it in Section 2.3.

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
        => -----------------------
                   P -- A? -> P' .

rl [par] :  P | Q -- A? -> P | Q'
        => ------------------------
                   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 2.3 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: one for the case when an action occurs in the lefthand side judgement,

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

and another one for the case when a metavariable occurs in the lefthand side judgement,

```
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 corresponding 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

```
op `[_=/=_`] : Act? Act? -> Judgement .
```

used to state constraints related with metavariables (which will be substituted by actions). This constraint is eliminated when it is fulfilled,

```
crl [dist] : [A =/= A'] => emptyJS  if A =/= A' .
```

where (normal) actions are used.

For the relabelling operator of CCS we have similar rewrite rules (see Appendix A).

Finally, process identifiers 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 the above 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 process `'b.0`) by rewriting the configuration composed of a judgement representing that transition:

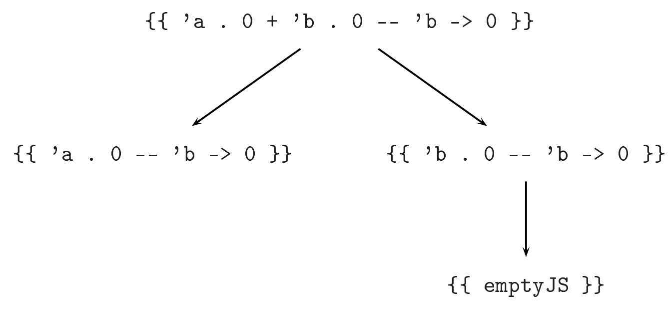
```
Maude> (rew {{ 'a . 'b . 0 -- 'a -> 'b . 0 }} .)
result Configuration : {{ emptyJS }}
```

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 the configuration `{{ '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, as shown in the following tree:



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

2.2 Maude's Metalevel

Rewriting logic is reflective [10, 11, 4], 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:

$$\mathcal{R} \vdash t \longrightarrow t' \quad \Leftrightarrow \quad \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

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` of terms;
- Maude modules are reified as terms in a data type `Module` of modules;
- the processes of reducing a term to normal form in a functional module and of finding whether such a normal form has a given sort are reified by a metalevel function `meta-reduce`;

```
op meta-reduce : Module Term -> Term [special ... ] .
```

- the process of applying a rule of a system module to a subject term at the top is reified by a metalevel function `meta-apply`;

```
op meta-apply : Module Term Substitution MachineInt ->
                ResultPair [special ... ] .
```

- the process of rewriting a term in a system module using Maude's default interpreter is reified by a metalevel function `meta-rewrite`;

```
op meta-rewrite : Module Term MachineInt -> Term
                  [special ... ] .
```

- parsing and pretty printing of a term in a signature, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also reified by corresponding metalevel functions.

We briefly summarize here how Maude terms are metarepresented as terms of sort `Term`; for a complete explanation we refer the reader to the Maude manual [8]. The signature for representing terms is declared in module `META-LEVEL` as follows:

```
subsort Qid < Term < TermList .
op {_}_ : Qid Qid -> Term .      *** constants
op _[_] : Qid TermList -> Term . *** ops. with arguments
op _,_ : TermList TermList -> TermList [assoc] . *** arguments
```

For example, the term `'a.0 + 'b.0`, of sort `Process` is metarepresented as

```
'_+_ [ '_._ [ { ''a } 'Qid , { '0 } 'Process ] ,
       '_._ [ { ''b } 'Qid , { '0 } 'Process ] ]
```

2.3 Searching in the tree of rewritings

In this section we show how the reflective properties of Maude [11] 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 the work in [1, 2, 7], 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* to specify the requeriments that the parameter must satisfy: it must have 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 metarepresented module `MOD` has an operation `ok` (defined at the object level, see Section 2.4), 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 the metalevel function `meta-apply`. The evaluation of `meta-apply(MOD,T,L,S,N)` applies (discarding the first `N` successful matches) a rule of module `MOD` with label `L`, partially instantiated with substitution `S`, to the term `T` (at the top level). It returns the resulting fully reduced term and the representation of the match used in the reduction, `{ T', S' }`, as a term of sort `ResultPair` built by means of the operator `{_,_}`.

In Section 2.1 we saw 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`. (However, the version given in Appendix A is completely general, for any number of new variables.) These variables are then substituted by new identifiers, which are quoted numbers.

```
 vars N M : MachineInt .
```

```
op new-var : MachineInt -> Term .
eq new-var(M) = {conc('' , index(' , M))}'Qid .
op createSubst : MachineInt -> Substitution .
eq createSubst(M) = (('NEW1@Qid <- new-var(M + 1));
                     ('NEW2@Qid <- new-var(M + 2));
                     ('NEW3@Qid <- new-var(M + 3))) .
```

The operations `conc` and `index` are defined in the module `QID`, and they are used to build quoted identifiers. For example, the evaluation of `new-var(5)` returns the metarepresentation `{''5}'Qid` of term `'5`. To avoid the clash of names of variables Full Maude renames them following the convention of adding the character '`@`' plus the name of its sort. Thus, for example, a variable `'NEW1` of sort `Qid` is renamed to `'NEW1@Qid`.

The signature for building substitutions is declared in module `META-LEVEL` as follows:

```
sorts Assignment Substitution .
subsort Assignment < Substitution .
op _<-_ : Qid Term -> Assignment .
op none : -> Substitution .
op _;_ : Substitution Substitution -> Substitution
         [assoc comm id: none] .
```

Then, the evaluation of `createSubst(5)` returns the substitution

```
'NEW1@Qid <- {''6}'Qid ; 'NEW2@Qid <- {''7}'Qid ;
'NEW3@Qid <- {''8}'Qid
```

used to replace the variables `NEW1`, `NEW2`, and `NEW3` with new quoted identifiers.

We define now a new operation `meta-apply'` which receives the greatest number `M` used to substitute variables in `T` and uses three new numbers to create three new (metarepresented) identifiers.

```
var L : Qid .
var T : Term .
var SB : Substitution .
op extTerm : ResultPair -> Term .
eq extTerm({T, SB}) = T .
op meta-apply' : Term Qid MachineInt MachineInt -> Term .
eq meta-apply'(T, L, N, M) = extTerm(meta-apply(MOD, T, L,
                                          createSubst(M), N)) .
```

The operation `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 [17] of a given term `T` 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 the module 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 rewArgs : 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
                        lowerRew(T, L, M),  *** rew. subterms
                        allRew(T, LS, M) .  *** rew. with labels LS
```

The evaluation of `topRew(T, L, N, M)` returns all the possible one-step rewritings at the top level of term `T` applying rule `L`, discarding the first `N` matches, and using numbers from `M+1` to create identifiers for new variables.

```
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 .
```

The evaluation of `lowerRew(T,L,M)` returns all the possible one-step rewritings of the subterms of term `T` applying rule `L`, and using numbers from `M+1` to create identifiers for new variables. If `T` is a constant (term without arguments), the empty list of terms is returned; otherwise, the operation `rewArgs` is used, which is defined recursively on its third argument, which represents the arguments of `T` not rewritten yet.

```
eq lowerRew({C}S, L, M) = ~ .
eq lowerRew(OP[TL], L, M) = rewArgs(OP, ~, TL, L, M) .
eq rewArgs(OP, Before, T, L, M) =
    rebuild(OP, Before, allRew(T, L, M), ~) .
eq rewArgs(OP, Before, (T, After), L, M) =
    rebuild(OP, Before, allRew(T, L, M), After) ,
    rewArgs(OP, (Before, T), After, L, M) .
```

The evaluation of `rebuild(OP, Before, TL, After)` returns all the terms of the form `OP[Before, T, After]` where `T` is a term in the term list `TL`. These built terms are metareduced before being returned.

```
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) .
```

For example, we can apply the operation `allRew` to the metarepresentation of the term `{{ 'a.0 + 'b.0 -- 'b -> 0 }}` to calculate all its rewritings, and we get the metarepresentation of the terms `{{ 'a.0 -- 'b -> 0 }}` and `{{ 'b.0 -- 'b -> 0 }}`.[3]

```
Maude> (red allRew( {{ 'a . 0 + 'b . 0 -- 'b -> 0 }}, labels, 0 ) .)
result TermList : {{ 'a . 0 -- 'b -> 0 }}, {{ 'b . 0 -- 'b -> 0 }}
```

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 .
```

[3]In Appendix B the reader can find these examples as they are introduced in the Maude system. Here we use $\overline{t}$ for the metarepresentation of term t.

2.4 Some examples

Now we can test the CCS semantics with some examples using different judgements. First, we specify a module `CCS-OK` extending the CCS syntax and semantic rules by defining some process constants to be used in the examples, defining the constant `context` used by the semantics, and also 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
  protecting CCS-SEMANTICS .
  ops p1 p2 p3 : -> Process .
  eq p1 = ('a . 0) + ('b . 0 | ('c . 0 + 'd . 0)) .
  eq p2 = ('a . 'b . 0 | (~ 'c . 0) ['a / 'c]) \ 'a .
  eq context = ('Proc =def 'a . tau . 'Proc) .
  eq p3 = ('Proc | ~ 'a . 'b . 0) \ '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 generic module `SEARCH`, we need the metarepresentation of module `CCS-OK`. We use the Full Maude `up` function [8, 13] to obtain the metarepresentation of a module or a term.

```
(mod META-CCS is
  including META-LEVEL .
  op METACCS : -> Module .
  eq METACCS = up(CCS-OK) .
endm)
```

We declare a view [8, 13] in order to instantiate the parameterized generic module `SEARCH` with it.

```
(view ModuleCSS 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`.

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

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

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

Process `p2` can perform action `tau` becoming `('b.0 | 0['a/'c])\'a`.

```
Maude> (red rewDepth({{ p2 -- tau -> ('b.0 | 0 ['a/'c])\'a }})).)
result Term : {{ emptyJS }}
```

In the same way, we can prove that process `p3` can perform action `tau`.

```
Maude> (red rewDepth({{ p3 -- tau -> (tau.'Proc | 'b.0)\'a }})).)
result Term : {{ emptyJS }}
```

In all these examples, we have had to provide the resulting process. In the positive proofs there is no problem (besides the cumbersome task of explicitly writing the resulting process), 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 Section 3.2 how this can be proved.

3 How to obtain new kinds of 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.

3.1 Including metavariables as processes

As we did with actions, we declare a new sort of metavariables as processes,

```
sort MetaVarProc .
op ?`(_`)P : Qid -> MetaVarProc .
```

add a new sort of possible processes,

```
sort Process? .
subsorts Process MetaVarProc < Process? .
var ?P : MetaVarProc .
var P? : Process? .
```

and modify the operation to build the basic transition judgements

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

For the prefix operator we have to add two new rules:

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

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

where, as in the case of metavariables as actions, we 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:

```
op `[_:=_`] : MetaVarProc Process? -> Judgement .
rl [bind] : {{ [?P := P] JS }} => {{ <proc ?P := P > JS }} .
op <proc_:=_>_ : MetaVarProc Process Process? -> Process?  .
op <proc_:=_>_ : MetaVarProc Process Judgement -> Judgement .
op <proc_:=_>_ : MetaVarProc Process JudgementSet -> JudgementSet .
```

For the rest of CCS operators, new rules have to be added to deal with metavariables in the second process of the transition judgement (see the complete set of rules in Appendix A).

3.2 More examples

Now, we can prove that process `p1` can perform action `'c`, by rewriting the judgement

```
p1 -- 'c -> ?('any)P,
```

where the metavariable `?('any)P` means that we do not care about the resulting process.

```
Maude> (red rewDepth({{ p1 -- 'c -> ?('any)P }}) .)
result Term : {{ emptyJS }}
```

We can also prove that process `p2` *cannot* perform action `'a`.

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

3.3 Successors of a process

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 have metavariables as processes, we can rewrite the transition judgement

```
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 keeping a set of bindings already produced in addition to the set of judgements to be reduced

```
op `{`{_|_`}`} : JudgementSet JudgementSet -> Configuration .
```

The bindings will be saved in the second argument by the `bind` rule:

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

We have to change also the function `ok`

```
vars JS JS' : JudgementSet .
eq ok({{ emptyJS | JS' }}) = solution .
ceq ok({{ JS | JS'}}) = maybe-sol if JS =/= emptyJS .
```

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 (in a depth-first way) the whole tree of rewritings finding *all* the nodes that satisfy the function `ok`. The operation `allSol`, added to the module `SEARCH`, returns a set with the terms representing 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 SEARCH-CCS, see module CCS-SUCC in Appendix A) an operation succ which, given the metarepresentation of a process and a set of metarepresentations of actions, returns the set of metarepresentations of the process successors.[4]

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

where filter (see Appendix A) is used to remove all the bindings involving metavariables different from ?('proc)P.

We can illustrate how these functions work with the processes defined in module CCS-OK in Section 2.4. We can calculate the successors of process p1 after performing action 'a or action 'b:

```
Maude> (red succ(p1, 'a U 'b) .)
result TermSet : 0 | ( 'c.0 + 'd.0 ) U 0
```

4 Extending the semantics to traces

In the CCS semantics we have used until now, only judgements of the form $P \xrightarrow{a} P'$ exist. We can extend it to sequences of actions or *traces*, that is, to judgements of the form $P \xrightarrow{a_1 a_2 \ldots a_n} P'$. The semantic rules defining these new transitions are

$$\frac{}{P \xrightarrow{\varepsilon} P} \qquad\qquad \frac{P \xrightarrow{a_1} P' \quad P' \xrightarrow{a_2 \ldots a_n} P''}{P \xrightarrow{a_1 a_2 \ldots a_n} P''}$$

where ε denotes the empty trace.

We can extend our framework by defining a new sort for traces

```
sort Trace .
subsort Act < Trace .
op nil : -> Trace .
op __ : Trace Trace -> Trace [assoc id: nil] .
var Tr : Trace .
```

and a new kind of judgements

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

Then, we can translate the above semantic rules to corresponding rewrite rules in an obvious way:

[4]In this definition we also use $\overline{t}$ for the metarepresentation of term t. The reader can find in Appendix A how these metarepresented terms are written in Maude.

```
rl [nil] :     P - nil -> P
        => ------------------
                emptyJS .

rl [seq] :             P - A Tr -> P'
        => ----------------------------------------
            P -- A -> ?(NEW1)P   ?(NEW1)P - Tr -> P' .
```

As we have done with judgements dealing with actions, we can also extend judgements dealing with traces in such a way that metavariables can be used as processes. We only have to modify the operation for building this kind of judgements

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

and add new rules for the case when a metavariable appears as the second process

```
rl [nil] :  P - nil -> ?P
        => ---------------
             [?P := P] .

rl [seq] :            P - A Tr -> ?P
        => -----------------------------------------
            P -- A -> ?(NEW1)P   ?(NEW1)P - Tr -> ?P .
```

5 Extension to weak transition semantics

Another important transition relation defined in CCS, $P \stackrel{a}{\Longrightarrow} P'$, does not observe τ transitions [20]. It is defined as

$$\frac{P\ (\stackrel{\tau}{\longrightarrow})^*\ Q \quad Q \stackrel{a}{\longrightarrow} Q' \quad Q'\ (\stackrel{\tau}{\longrightarrow})^*\ P'}{P \stackrel{a}{\Longrightarrow} P'}$$

where $(\stackrel{\tau}{\longrightarrow})^*$ denotes the reflexive, transitive closure of $\stackrel{\tau}{\longrightarrow}$, and it is defined in the following way for a general action a

$$\frac{}{P\ (\stackrel{a}{\longrightarrow})^*\ P} \qquad\qquad \frac{P \stackrel{a}{\longrightarrow} P' \quad P'\ (\stackrel{a}{\longrightarrow})^*\ P''}{P\ (\stackrel{a}{\longrightarrow})^*\ P''}$$

The weak transition semantics is extended to traces as follows:

$$\frac{P\ (\stackrel{\tau}{\longrightarrow})^*\ P'}{P \stackrel{\varepsilon}{\Longrightarrow} P'} \qquad\qquad \frac{P \stackrel{a_1}{\Longrightarrow} P' \quad P' \stackrel{a_2 \ldots a_n}{\Longrightarrow} P''}{P \stackrel{a_1 a_2 \ldots a_n}{\Longrightarrow} P''}$$

5.1 Defining the extensions in Maude

We show in this section how these extensions can be defined in Maude. First, we define the reflexive, transitive closure of the basic transition relation we have already implemented. We need an operator to build the new kind of judgements

```
op _`(--_->`)*_ : Process? Act Process? -> Judgement [prec 50] .
```

where we allow metavariables as processes because we need them for the implementation of the weak transition relation. The rewrite rules defining the closure are as follows:

```
rl [refl] :   P (-- A ->)* P
         => --------------------
                 emptyJS .

rl [refl] :  P (-- A ->)* ?P
         => -----------------
                [?P := P] .

rl [tran] :             P (-- A ->)* P?
         => ----------------------------------------------
             P -- A -> ?(NEW1)P   ?(NEW1)P (-- A ->)* P? .
```

In the same way, we can define in Maude the weak transition relation both for actions and for traces:

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

rl [act] :                 P == A ==> P?
        => ------------------------------------------------------
            P (-- tau ->)* ?(NEW1)P   ?(NEW1)P -- A -> ?(NEW2)P
                       ?(NEW2)P (-- tau ->)* P? .

op _=_==>_ : Process? Trace Process? -> Judgement [prec 50] .

rl [nil] :    P = nil ==> P?
        => --------------------
            P (-- tau ->)* P? .

rl [seq] :                 P = A Tr ==> P?
        => --------------------------------------------
            P == A ==> ?(NEW1)P   ?(NEW1)P = Tr ==> P? .
```

And in the same way as before, we can define a function to calculate the successors of a process with respect to the weak transition semantics:

$$wsucc(P, As) = \{P' \mid P \stackrel{a}{\Longrightarrow} P' \wedge a \in As\}.$$

```
op wsucc : Term TermSet -> TermSet .
eq wsucc(T, {}) = {} .
eq wsucc(T, A U AS) = filter(
     allSol({{ T == A ==> ?('proc)P | emptyJS }}), ?('proc)P)
    U wsucc(T, AS) .
```

5.2 Examples

By continuing with our running example of module `CCS-OK` in Section 2.4, we can now prove that process `p2` can perform an observable `'b` (and we do not care about which process it becomes):

```
Maude> (red rewDepth({{ p2 == 'b ==> ?('any)P }}) .)
result Term : {{ emptyJS }}
```

We can prove that the recursive process `'Proc` can execute successively three observable `'a`'s and become the same process:

```
Maude> (red rewDepth({{ 'Proc = 'a 'a 'a ==> 'Proc }}) .)
result Term : {{ emptyJS }}
```

Finally, we can calculate the successors of the recursively defined process `'Proc` after performing action `'a` allowing τ transitions:

```
Maude> (red wsucc('Proc,'a) .)
result TermSet : tau . 'Proc U 'Proc
```

6 Modal logic for CCS processes in Maude

We show now how we can define in Maude the semantics of a modal logic for CCS processes by using the operations of previous sections.

6.1 Hennessy-Milner logic

We introduce a *modal* logic for describing local capabilities of CCS processes that we will implement in Maude in the next section. This logic is a version of Hennessy-Milner logic [14] and its semantics is presented in [24]. Formulas are as follows

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

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

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

6.2 Implementation in Maude

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

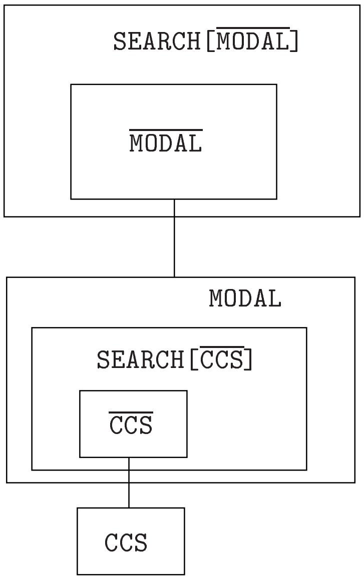
```
(mod MODAL-LOGIC is
  protecting CCS-SUCC .
  sort HMFormula .
  ops tt ff : -> HMFormula .
  op _/\_ : HMFormula HMFormula -> HMFormula .
  op _\/_ : HMFormula HMFormula -> HMFormula .
  op `[_`]_ : TermSet HMFormula -> HMFormula .
  op <_>_ : TermSet HMFormula -> HMFormula .
```

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 .
endm)
```

These rules are also nondeterministic. For example, the application of the two rules `or` is nondeterministic because they have the same lefthand side, and the rule `ex` is also nondeterministic because of multiple matchings modulo associativity and commutativity.

We can instantiate the module `SEARCH` in Section 2.3 with the metarepresentation of the module containing the definition of the modal logic semantics. So we obtain the following structure of modules:



6.3 Examples

As an example taken from [24], we show some modal formulas satisfied by a vending machine `'Ven` 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. We can prove that `'Ven |= ['big, 'little]ff` in Maude:

```
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
```

`'Ven` also satisfies that when a coin has been deposited no other coin can be inserted,

```
Maude> (red rewDepth('Ven |= [ '1p U '2p ] [ '1p U '2p ] ff) .)
```

```
result Term : emptyJS
```

Finally, after a coin is deposited and a button is depressed, an item (big or little) can be collected,

```
Maude> (red rewDepth(
   'Ven |= ['1p U '2p]['big U 'little]<'collectB U 'collectL>tt) .)
```

```
result Term : emptyJS
```

6.4 More modalities

If we want to give to the (silent) τ action a special status, we can introduce new modalities $[\![K]\!]$ and $\langle\!\langle K\rangle\!\rangle$ defined by means of the weak transition relation

$$
\begin{array}{ll}
P \models [\![K]\!]\Phi & \text{iff } \forall\, Q \in \{P' \mid P \overset{a}{\Longrightarrow} P' \wedge a \in K\} \,.\, Q \models \Phi \\
P \models \langle\!\langle K\rangle\!\rangle\Phi & \text{iff } \exists\, Q \in \{P' \mid P \overset{a}{\Longrightarrow} P' \wedge a \in K\} \,.\, Q \models \Phi
\end{array}
$$

The implementation in Maude of these new modalities is as follows:

```
op `[`[_`]`]_ : TermSet HMFormula -> HMFormula .
op <<_>>_ : TermSet HMFormula -> HMFormula .
rl [box]  : P |= [[ K ]] Phi => forall(wsucc(P, K), Phi) .
rl [diam] : P |= << K >> Phi => exists(wsucc(P, K), Phi) .
```

where we use the operation `wsucc` to calculate the successors of process `P` with respect to the weak transition semantics.

We can now prove some properties of a railroad crossing system specified in a CCS context 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 process `'Crossing` satisfies that when a car and a train arrive to the crossing, only one of them has the possibility to cross it.

```
Maude> (red RewDepth(
   'Crossing |= [['car]][['train]]((<<~ 'ccross>> tt)
                                   \/ (<<~ 'tcross>> tt))) .)
result Term : emptyJS
Maude> (red RewDepth(
   'Crossing |= [['car]][['train]]((<<~ 'ccross>> tt)
                                   /\ (<<~ 'tcross>> tt))) .)
result Term : error*
```

7 Comparison with Isabelle

In this section we present how the CCS semantics and the Hennessy-Milner modal logic can be represented and reasoned about within the theorem prover Isabelle [21], and compare this framework with ours.

Isabelle is a generic system for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, Higher-Order Logic [21].

Working with Isabelle means creating theories, which are named collection of types, functions, and theorems. HOL contains a theory *Main*, which is the union of all the basic predefined theories like arithmetic, lists, sets, etc. Our theories include this one as a parent theory. In a theory there are types, terms, and formulas of HOL. Terms are formed as in functional programming by applying functions to arguments, and formulas are terms of type `bool`.

Isabelle distinguishes free and bound variables as usual. In addition, Isabelle has a third kind of variable, called a *schematic variable* or *unknown*, which starts with a `?`. Logically, an unknown is a free variable, but it may be instantiated (through unification) to another term during the proof process. Unknowns correspond to what we have called *metavariables*.

Inductive datatypes are part of almost every non-trivial application of HOL. They are introduced by means of the keyword **datatype**, and defined by means of constructors. For example, CCS processes are naturally modelled as the following datatype.

```
datatype process = Stop                      ("0")
       | Prefix action  process     ("_ . _" [120, 110] 110)
       | Sum    process process     (infixl "+" 85)
       | Par    process process     (infixl "|" 90)
       | Rel    process label label ("_ [ _ =: _ ]" [100,120,120] 100)
       | Rest   process label       ("_ \ _" [100, 120] 100)
       | Id id
```

Functions (introduced with keyword **consts**) on datatypes are usually defined by *primitive recursion*. The keyword **primrec** is followed by a list of equations. Using these tools we define the CCS contexts (see Appendix C).

In order to define the CCS semantics in Isabelle/HOL, we define in an inductive way a set *trans* of triples *(P,a,P')*. We use the usual mixfix notation *P - a → P'* to represent that *(P,a,P') ∈ trans*. Inductiveness means that the set of transitions consists exactly of those triples *(P,a,P')* that can be derived from the set of transition rules. Inductively defined sets are introduced in Isabelle by means of an **inductive** declaration, which consists of introduction rules. These rules are straightforward translations of the CCS semantics rules. An inference rule of the form

$$\frac{\phi_1 \ \cdots \ \phi_n}{\phi}$$

is formalized in Isabelle's meta-logic as the axiom $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$.

Although we do not use unknowns directly in the rules, when constructing a theory Isabelle puts the rules into a standard form with all free variables converted into schematic ones, which let unification instantiate the rule.

These are some of the rules defining the CCS semantics (see complete code in Appendix C):

```
inductive trans
intros
  prefi: "A . P - A → P"
  sum1:  "P1 - A → P ⟹ P1 + P2 - A → P"
  sum2:  "P2 - A → P ⟹ P1 + P2 - A → P"
  par1:  "P1 - A → P ⟹ P1 | P2 - A → P | P2"
  par2:  "P2 - A → P ⟹ P1 | P2 - A → P1 | P"
  par3:  "⟦ P1 - va L → P ; P2 - va (compl L) → Q ⟧
```

```
  ⟹  P1 | P2 - tau → P | Q"
```

Simplification is one of the central theorem proving tools in Isabelle and many other systems. In its most basic form, simplification means repeated application of equations from left to right (term rewriting). To facilitate simplification, theorems can be declared to be simplification rules with the attribute `[simp]`, in which case proofs by simplification make use of these rules automatically. In addition the constructs **datatype** and **primrec** invisibly declare useful simplification rules.

Isabelle proofs normally use *resolution* to support backward proof, where we start with a goal and refine it to progressively simpler subgoals until all have been solved. Isabelle's *classical reasoner* is a family of tools that perform proofs automatically. We can extend the scope of the classical reasoner by giving it new rules (`intro:`). Classical reasoner uses search and backtracking in order to *prove* a goal. We use the `force` method, that applies the classical reasoner and simplifier to one goal.

As we did with Maude (Section 2.4), we can test the CCS semantics with some examples. Process constants are declared by means of definitions (**defs**), which are intended to express abbreviations, and are also declared to be simplification rules.

```
consts p1 :: process
defs p1_def[simp]: "p1 ≡ ( va (nm ''a'') . 0) +
     (va (nm ''b'') . 0 | (va (nm ''c'') . 0 + va (nm ''d'') . 0))"

consts p2 :: process
defs p2_def[simp]: "p2 ≡ ( va (nm ''a'') . va (nm ''b'') . 0 |
     ( va (conm ''c'') . 0) [ nm ''a'' =: nm ''c''] ) \ nm ''a''"

lemma "p1 - va (nm ''c'') → va (nm ''b'') . 0 | 0"
by (force intro: trans.intros)
```

We can also prove that a process can perform a given action, using an unknown `?P`, as we did in Section 3.2.

```
lemma "p2 - tau  → ?P"
by (force intro: trans.intros)
```

But in order to prove that a given transition is *not* possible, we cannot simply use one of the automatic proof methods; instead, we have to use induction on the relation `trans`. Isabelle generates an elimination rule for case analysis (`cases` rule) and an induction rule associated with each inductively defined set.

```
lemma "¬(p2 - va (nm ''a'') → ?P)"
apply clarsimp
by (erule trans.cases, clarsimp+)+
```

We can define the modal logic syntax and semantics as we have done for CCS. We use a datatype `formula` and an inductively defined set `sat` representing the satisfaction relation.

```
inductive sat
 intros
  ttR[intro!]:   "P ⊨ tt"
  andR:  "⟦ P ⊨ Phi ; P ⊨ Psi ⟧ ⟹ P ⊨ (Phi ∧ Psi)"
  orR1:  "P ⊨ Phi ⟹  P ⊨ (Phi ∨ Psi)"
  orR2:  "P ⊨ Psi ⟹  P ⊨ (Phi ∨ Psi)"
  diamR: "⟦ P - A → Q ; A ∈ AS; Q ⊨ Phi ⟧ ⟹ P ⊨ ⟨ AS ⟩ Phi"
  boxR:  "(∀ Q A . ( (P - A → Q ∧ A ∈ AS) ⟶ Q ⊨ Phi)) ⟹ P ⊨ [ AS ] Phi"
```

In order to prove that a given process satisfies a formula, we have to use induction on the relation `sat`. This is due to the universal quantifier in the `boxR` rule, which refers to all the successors of a process. The `ind_cases` method applies an instance of the `cases` rule for the supplied pattern.

```
lemma "(va (nm ''a'') . va (nm ''b'') . 0)
       ⊨ [ {va (nm ''a'')} ]( ⟨{va (nm ''b'')}⟩ tt)"
apply (rule sat.boxR, clarify)
apply (ind_cases "A . P - B → Q")
by (force intro: trans.intros sat.intros)
```

7.1 Comparison with the Maude representation

Isabelle has been designed as a logical framework and theorem prover. Hence, the concepts of unknowns and unification are basic, in such a way that the user does not have to worry about them. On the contrary, Maude does not directly support unknowns. But, as we have seen, we can represent them within Maude, and due to its reflective capabilities, deal with them creating new fresh metavariables, and propagating their values when they become known.

Isabelle/HOL also has several automatic tools that help prove theorems. For example, as we have seen above, Isabelle automatically generates an induction rule when an inductive set is defined. Obviously, this adds a lot of power. But the user must know how to use all these tools, that is, when to use introduction or elimination rules, or when to use induction, for example. In Maude, we only use the rewrite rules that define the semantics and the exhaustive search strategy that (blindly) uses them, being able to prove both sentences about CCS and the modal logic in the same way.

Isabelle, as a logical framework, 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.

Moreover, rewriting logic is as a coin with two sides, where deduction can also be interpreted as computation. This allows us to obtain information from proofs, as we did with the operation `succ` of process successors. Although we can represent in Isabelle the set of successors of a process, and prove whether a given process is in this set, we cannot compute (or show) this set.

8 LOTOS symbolic semantics in Maude

We have use the ideas presented in this paper to represent in Maude a symbolic semantics for LOTOS [3]. Since this semantics is more complicated than the CCS one, and LOTOS specifications are quite more complicated than simple CCS processes, we have tried to improve the ideas presented above in several directions, that we are going to comment now.

First, in order to present the semantics rules in a more elegant way, we write the rules for each LOTOS operator assuming metavariables in every place where a metavariable may appear (within a transition), instead of what we have done for CCS where we have written several rules depending on the places where metavariables appear. Then, we have rules that reduce other kinds of transitions (without metavariables everywhere) to the above ones.

Continuing with the idea of presenting the semantics in a way as general as possible, we use an operator `_[[_/_]]` to represent a syntactic substitution. It is used when the binding of a metavariable to a concrete value has to be propagated to the rest of judgements. The improvement is that we can define the behaviour of this operator at the metalevel only *once*, instead of using several overloaded auxiliary functions as we have done in Sections 2.1 and 3.1.

Regarding efficiency, we have changed the sets of judgements for *sequences* of judgements in order to avoid multiple matchings modulo commutativity. This means that judgements will be ordered, and they will be proved form left to right. We have also changed the search strategy, which now tries to rewrite 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. 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.

We have also integrated the LOTOS tool with Full Maude (which is highly extensible [13]), adding the capability of introducing LOTOS specifications to Maude, and working with them by means of specific new commands. In this way, the tool user does not have to know all the details of the implementation, especially the metalevel ones.

9 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`. In particular, we have used metavariables (together with the substitution cabability of the metalevel operation `meta-apply`) to solve the problem of new variables, and a search strategy which deals with conceptual trees of rewritings to solve the problem of nondeterminism.

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 extending the semantics to traces, defining weak transition semantics, and answering which are the successors of a process after performing an action. In its turn, this allows a representation of the semantics of the Hennessy-Milner modal logic in a quite similar way as it is done in its mathematical definition.

We have compared our approach with the Isabelle/HOL one, concluding that although

Maude, being a so general framework, is not so easy to use, it can be used for applying higher-order techniques in a first-order framework by means of reflection.

Other specific-purpose tools, such as the Concurrency Workbench [12], 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 LOTOS. We are building a tool where we can reason about complete LOTOS specifications with both data and behaviour. By translating the ACT ONE specifications to Maude functional modules, we obtain an execution environment for Full LOTOS.

Acknowledgements

We are very grateful to José Meseguer and the FORTE/PSTV 2000 referees for their very helpful comments on earlier versions of this paper. We would like to thank David Basin for his very fruitful comments about the Isabelle/HOL foundations.

This paper is a longer and revised version of [25].

References

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

[2] R. Bruni, J. Meseguer, and U. Montanari. Internal strategies in a rewriting implementation of tile systems. In Kirchner and Kirchner [15]. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[3] 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*, pages 184–200. Kluwer Academic Publishers, 2001.

[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. Quesada. Maude as a metalanguage. In Kirchner and Kirchner [15]. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *A Maude Tutorial*. SRI International, Mar. 2000. `http://maude.csl.sri.com`.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Using Maude. In T. Maibaum, editor, *Proc. Third Int. Conf. Fundamental Approaches to Software Engineering, FASE 2000, Berlin, Germany, March/April 2000*, LNCS 1783, pages 371–374. Springer, 2000.

[8] 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.

[9] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods, Proc. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Volume II*, LNCS 1709, pages 1684–1703. Springer, 1999.

[10] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96*, pages 263–288, 1996.

[11] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications, Asilomar, California, U.S.A*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Sept. 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[12] 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.

[13] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, 1999.

[14] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, Jan. 1985. `http://www.acm.org/pubs/toc/Abstracts/0004-5411/2460.html`.

[15] C. Kirchner and H. Kirchner, editors. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, Nancy, France*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[16] 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.

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

[18] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, NATO ASI Series F: Computer and Systems Sciences 165, pages 347–398. Springer, 1998.

[19] J. Meseguer, K. Futatsugi, and T. Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. In *Proc. of the 1992 International Symposium on New Models for Software Architecture, Tokyo, Japan, November 1992*, pages 61–106. Research Institute of Software Engineering, 1992.

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

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

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

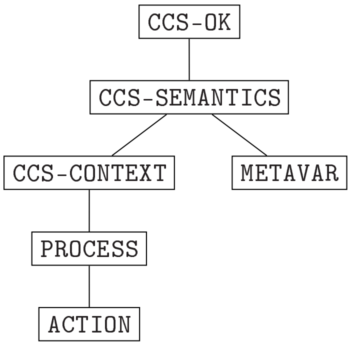
[23] M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic. In *Proc. of LFM'99: Workshop on Logical Frameworks and Meta-Languages*, Paris, France, Sept. 1999.

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

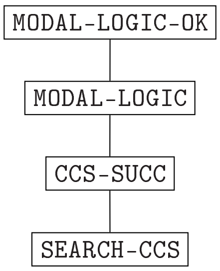
[25] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In T. Bolognesi and D. Latella, editors, *Formal Methods For Distributed System Development. FORTE/PSTV 2000*, pages 351–366. Kluwer Academic Publishers, 2000.

A Complete Maude specification

In order to help to understand the module structure we first present two diagrams showing module importation. The modules used to represent the CCS semantics are as follows:



The module `CCS-OK` (metarepresented) is used to instantiate the generic module `SEARCH`, and this instantiation is in module `SEARCH-CCS`. The modules used to represent the modal logic are as follows:



The module `MODAL-LOGIC-OK` (metarepresented) is used to instantiate again the generic module `SEARCH`, and this instantiation is in `SEARCH-MODAL-CCS`.

The complete Maude specification is the following one:

```
--- an action is the silent action or a label
(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)

--- CCS syntax
(fmod PROCESS is
  protecting ACTION .
  sorts ProcessId Process .
  subsorts 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)

--- defining equations and contexts
(fmod CCS-CONTEXT is
  protecting PROCESS .
  sorts BadProcess Context BadContext .
  subsort Process < BadProcess .
  subsort Context < BadContext .
  op _=def_ : ProcessId Process -> Context [prec 40] .
  op nil : -> Context .
  op _&_ : BadContext BadContext -> BadContext
             [assoc comm id: nil prec 42] .
  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)) = P .
  ceq def(X, ((X' =def P) & C)) = def(X, C) if X =/= X' .
endfm)

(fmod METAVAR is
  protecting QID .
  sorts MetaVarAct MetaVarProc .
```

```
  op ?`(_`)A : Qid -> MetaVarAct .
  op ?`(_`)P : Qid -> MetaVarProc .
endfm)

--- CCS operational semantics
(mod CCS-SEMANTICS is
  pr CCS-CONTEXT .  pr METAVAR .

  sort Act? .
  subsort Act < Act? .
  subsort MetaVarAct < Act? .
  sort Process? .
  subsort Process < Process? .
  subsort MetaVarProc < Process? .

  vars L L' M : Label .
  vars A A' A'' : Act .
  vars A? A'? : Act? .
  vars P P' Q Q' : Process .
  vars P? P'? Q? Q'? : Process? .
  var X : ProcessId .
  vars ?A ?A' : MetaVarAct .
  vars ?P ?Q ?R : MetaVarProc .

  vars NEW1 NEW2 NEW3 : Qid .
  op how-many-mv : -> MachineInt .  *** number of metavariables used
  eq how-many-mv = 3 .

  op ~_ : Act? -> Act? .
  op _\_ : Process? Label -> Process? .
  op _`[_/_`] : Process? Label Label -> Process? .
  op _|_ : Process? Process? -> Process? .

  sort Judgement .
  op _--_->_ : Process? Act? Process? -> Judgement [prec 50] .
  op `[_:=_`] : MetaVarAct Act -> Judgement .
  op `[_=/=_`] : Act? Act? -> Judgement .
  op `[_:=_`] : MetaVarProc Process? -> Judgement .

  sort JudgementSet .
  op emptyJS : -> JudgementSet .
  subsort Judgement < JudgementSet .
  op __ : JudgementSet JudgementSet -> JudgementSet
         [assoc id: emptyJS prec 60] .
  var J : Judgement .
  vars JS JS' : JudgementSet .

  *** commutativity
```

```
eq (P? -- A? -> Q?) [?A := A] = [?A := A] (P? -- A? -> Q?) .
eq (P? -- A? -> Q?) [?P := P] = [?P := P] (P? -- A? -> Q?) .


sort Configuration .
op `{`{_`}`} : JudgementSet -> Configuration  .
op `{`{_|_`}`} : JudgementSet JudgementSet -> Configuration  .

*** Assignments MetaVarAct := Act
op <act_:=_>_ : MetaVarAct Act Act? -> Act? .
op <act_:=_>_ : MetaVarAct Act Judgement -> Judgement .
op <act_:=_>_ : MetaVarAct Act JudgementSet -> JudgementSet .
eq <act ?A := A > ?A = A .
ceq <act ?A := A > ?A' = ?A' if ?A =/= ?A' .
eq <act ?A := A > A' = A' .
eq <act ?A := A > (~ A?) = ~(<act ?A := A > A?) .
eq <act ?A := A > (Q? -- A? -> Q'?) =
   (Q? -- <act ?A := A > A? -> Q'?) .
eq <act ?A := A > [ ?A' := A' ] = [?A' :=  A'] .
eq <act ?A := A > [ A? =/= A'? ] =
   [<act ?A := A > A? =/= <act ?A := A > A'?] .
eq <act ?A := A > [ ?P := P'? ] = [ ?P := P'? ] .
eq <act ?A := A > emptyJS = emptyJS .
ceq <act ?A := A > (J JS) = (<act ?A := A > J) (<act ?A := A > JS)
                         if JS =/= emptyJS .

*** Assignments MetaVarProc := Process
op <proc_:=_>_ : MetaVarProc Process Process? -> Process? .
op <proc_:=_>_ : MetaVarProc Process Judgement -> Judgement .
op <proc_:=_>_ : MetaVarProc Process JudgementSet ->
                 JudgementSet .
eq <proc ?P := P > ?P = P .
ceq <proc ?P := P > ?Q = ?Q if ?P =/= ?Q .
eq <proc ?P := P > P' = P' .
eq <proc ?P := P > (Q? \ L) = (<proc ?P := P > Q?) \ L .
eq <proc ?P := P > (Q? [ M / L ]) =
     (<proc ?P := P > Q?) [ M / L ] .
eq <proc ?P := P > (P? | Q?) =
     (<proc ?P := P > P?) | (<proc ?P := P > Q?) .
eq <proc ?P := P > (Q? -- A? -> Q'?) =
     ((<proc ?P := P > Q?) -- A? -> (<proc ?P := P > Q'?)) .
eq <proc ?P := P > [ ?A := A ] = [ ?A := A ] .
eq <proc ?P := P > [ A? =/= A'? ] = [ A? =/= A'? ] .
eq <proc ?P := P > [ ?Q := Q? ] = [ ?Q := <proc ?P := P > Q? ] .
eq <proc ?P := P > emptyJS = emptyJS .
ceq <proc ?P := P > (J JS) = (<proc ?P := P > J)
     (<proc ?P := P > JS) if JS =/= emptyJS .
```

```
rl [bind] : {{ [?A := A] JS }} => {{ <act ?A := A > JS }} .
rl [bind] : {{ [?A := A] JS | JS' }} =>
            {{ (<act ?A := A > JS) | [?A := A] JS' }} .
crl [dist] : [A =/= A'] => emptyJS  if A =/= A' .
rl [bind] : {{ [?P := P] JS }} => {{ <proc ?P := P > JS }} .
rl [bind] : {{ [?P := P] JS | JS' }} =>
            {{ (<proc ?P := P > JS) | [?P := P] JS' }} .

*** Prefix

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

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

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

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

*** Summation

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

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

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

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

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

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

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

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


*** Composition

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

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

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

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

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] .

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

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

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

```
        => ----------------------------------------------
            Q -- A -> ?(NEW1)P  [?Q := P | ?(NEW1)P] .

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

rl [par] :                 P | Q -- tau -> ?R
     => -------------------------------------------------------
         P -- ?(NEW1)A -> ?(NEW2)P   Q -- ~ ?(NEW1)A -> ?(NEW3)P
                   [?R := ?(NEW2)P | ?(NEW3)P] .

rl [par] :                  P | Q -- ?A -> ?R
     => -------------------------------------------------------
         P -- ?(NEW1)A -> ?(NEW2)P   Q -- ~ ?(NEW1)A -> ?(NEW3)P
               [?A := tau] [?R := ?(NEW2)P | ?(NEW3)P] .

*** Restriction

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] .

crl [res] :           P \ L -- A -> ?P
       => -------------------------------------------
           P -- A -> ?(NEW1)P [?P := ?(NEW1)P \ L]
       if (A =/= L) and (A =/= ~ L) .

rl [res] :           P \ L -- ?A -> ?P
       => --------------------------------------------
           P -- ?A -> ?(NEW1)P  [?A =/= L] [?A =/= ~ L]
                    [?P := ?(NEW1)P \ L] .

*** Relabelling

rl [rel] :  P[M / L] -- M -> P'[M / L]
        => -----------------------------
                   P -- L -> P' .

rl [rel] :  P[M / L] -- ~ M -> P'[M / L]
        => -------------------------------
                  P -- ~ L -> P' .
```

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

rl [rel] :  P[M / L] -- ?A -> P'[M / L]
        => ----------------------------
              (P -- L -> P') [?A := M] .

rl [rel] :  P[M / L] -- ?A -> P'[M / L]
        => ------------------------------
            P -- ~ L -> P'  [?A := ~ M] .

rl [rel] :      P[M / L] -- ?A -> P'[M / L]
        => -----------------------------------------
             P -- ?A -> P' [?A =/= L] [?A =/= ~ L] .

rl [rel] :             P[M / L] -- M -> ?P
        => ------------------------------------------------
            P -- L -> ?(NEW1)P [?P := ?(NEW1)P [ M / L ]] .

rl [rel] :              P[M / L] -- ~ M -> ?P
        => ----------------------------------------------------
            P -- ~ L -> ?(NEW1)P [?P := ?(NEW1)P [ M / L ]]  .

rl [rel] :              P[M / L] -- ?A -> ?P
     => --------------------------------------------------------
         P -- L -> ?(NEW1)P [?A := M] [?P := ?(NEW1)P [ M / L ]] .

rl [rel] :           P[M / L] -- ?A -> ?P
        => --------------------------------------------
                     P -- ~ L -> ?(NEW1)P
              [?A := ~ M] [?P := ?(NEW1)P [ M / L ]] .

crl [rel] :               P[M / L] -- A -> ?P
        => ------------------------------------------------
             P -- A -> ?(NEW1)P [?P := ?(NEW1)P [ M / L ]]
        if (A =/= L) and (A =/= ~ L) .

rl [rel] :             P[M / L] -- ?A -> ?P
        => --------------------------------------------
            P -- ?A -> ?(NEW1)P [?A =/= L] [?A =/= ~ L]
                   [?P := ?(NEW1)P [ M / L ]] .

*** Definition

crl [def] :         X -- A -> P'
        => ----------------------------------
```

```
              def(X, context) -- A -> P'
          if (X definedIn context) .

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

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

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

*** Extension to traces of actions
  sort Trace .
  subsort Act < Trace .
  op nil : -> Trace .
  op __ : Trace Trace -> Trace [assoc id: nil] .
  op _-_->_ : Process? Trace Process? -> Judgement [prec 50] .
  var Tr : Trace .

  eq (P? - Tr -> Q?) [?A := A] = [?A := A] (P? - Tr -> Q?) .
  eq (P? - Tr -> Q?) [?P := P] = [?P := P] (P? - Tr -> Q?) .


  rl [nil] :     P - nil -> P
         => -------------------
             emptyJS .

  rl [nil] :   P - nil -> ?P
         => ------------------
               [?P := P] .

  rl [seq] :             P - A Tr -> P'
         => -----------------------------------------
             P -- A -> ?(NEW1)P   ?(NEW1)P - Tr -> P' .

  rl [seq] :             P - A Tr -> ?P
         => -----------------------------------------
             P -- A -> ?(NEW1)P   ?(NEW1)P - Tr -> ?P .

  eq <act ?A := A > (Q? - Tr -> Q'?) = (Q? - Tr -> Q'?)  .
  eq <proc ?P := P > (Q? - Tr -> Q'?) =
```

```
    ((<proc ?P := P > Q?) - Tr -> (<proc ?P := P > Q'?))  .


*** Extension to reflexive, transitive closure    (-- A ->)*
  op _`(--_->`)*_ : Process? Act Process? -> Judgement [prec 50] .

  eq (P? (-- A ->)* Q?) [?A := A'] = [?A := A'] (P? (-- A ->)* Q?) .
  eq (P? (-- A ->)* Q?) [?P := P] = [?P := P] (P? (-- A ->)* Q?) .

  rl [refl] :   P (-- A ->)* P
          => --------------------
              emptyJS .

  rl [refl] :  P (-- A ->)* ?P
          => -----------------
                 [?P := P] .

  rl [tran] :             P (-- A ->)* P'
          => ---------------------------------------------
              P -- A -> ?(NEW1)P   ?(NEW1)P (-- A ->)* P' .

  rl [tran] :             P (-- A ->)* ?P
          => ---------------------------------------------
              P -- A -> ?(NEW1)P   ?(NEW1)P (-- A ->)* ?P .

  eq <act ?A := A > (P? (-- A' ->)* P'?) = (P? (-- A' ->)* P'?) .
  eq <proc ?P := P > (Q? (-- A ->)* Q'?) =
     ((<proc ?P := P > Q?) (-- A ->)* (<proc ?P := P > Q'?)) .

*** Extension to double arrow   == A =>
  op _==_==>_ : Process? Act Process? -> Judgement [prec 50 ] .

  eq (P? == A? ==> Q?) [?A := A] = [?A := A] (P? == A? ==> Q?) .
  eq (P? == A? ==> Q?) [?P := P] = [?P := P] (P? == A? ==> Q?) .

  rl [act] :                 P == A ==> P'
         => ---------------------------------------------------
             P (-- tau ->)* ?(NEW1)P   ?(NEW1)P -- A -> ?(NEW2)P
                        ?(NEW2)P (-- tau ->)* P' .

  rl [act] :                 P == A ==> ?P
         => ---------------------------------------------------
             P (-- tau ->)* ?(NEW1)P   ?(NEW1)P -- A -> ?(NEW2)P
                        ?(NEW2)P (-- tau ->)* ?P .

  eq <act ?A := A > (P? == A' ==> P'?) = (P? == A' ==> P'?) .
  eq <proc ?P := P > (Q? == A ==> Q'?) =
     ((<proc ?P := P > Q?) == A ==> (<proc ?P := P > Q'?)) .
```

```
*** Extension to double arrow with traces  = Tr ==>
  op _=_==>_ : Process? Trace Process? -> Judgement [prec 50] .

  eq (P? = A? ==> Q?) [?A := A] = [?A := A] (P? = A? ==> Q?) .
  eq (P? = A? ==> Q?) [?P := P] = [?P := P] (P? = A? ==> Q?) .

  rl [nil] :    P = nil ==> P'
          => --------------------
              P (-- tau ->)* P' .

  rl [nil] :    P = nil ==> ?P
          => ----------------------
              P (-- tau ->)* ?P .

  rl [seq] :             P = A Tr ==> P'
          => -------------------------------------------
              P == A ==> ?(NEW1)P   ?(NEW1)P = Tr ==> P' .

  rl [seq] :             P = A Tr ==> ?P
          => -------------------------------------------
              P == A ==> ?(NEW1)P   ?(NEW1)P = Tr ==> ?P .

  eq <act ?A := A > (P? = Tr ==> P'?) = (P? = Tr ==> P'?) .
  eq <proc ?P := P > (Q? = Tr ==> Q'?) =
     ((<proc ?P := P > Q?) = Tr ==> (<proc ?P := P > Q'?)) .
endm)

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

(fmod SEARCH[M :: AMODULE] is
  sort TermSet .
  subsort Term < TermSet .
  sorts Pair PairSeq .
  subsort Pair < PairSeq .
  vars F C S OP OP' : Qid .
  vars T T' : Term .
  vars TL Before After : TermList .
  vars M N : MachineInt .
  var L : Qid .
  var LS : QidList .
  var SB : Substitution .
  var TS : TermSet .
```

```
var PS : PairSeq .

*** Extension of TermList with an identity element
op ~ : -> TermList .
eq ~, TL = TL .
eq TL, ~ = TL .

op extTerm : ResultPair -> Term .
op extSubst : ResultPair -> Substitution .
eq extTerm({T, SB}) = T .
eq extSubst({T, SB}) = SB .

*** creation of new metavariables
op new-var : MachineInt -> Term .
eq new-var(N) = {conc('' , index(' , N))}'Qid .
op createSubst : MachineInt MachineInt -> Substitution .
eq createSubst(0, M) = none .
ceq createSubst(N, M) = (createSubst(_-_(N,1), M) ;
                        (conc('NEW , conc(index(' , N), '@Qid))
                         <- new-var(M + N))) if N =/= 0 .

op meta-apply' : Term Qid MachineInt MachineInt -> Term .
op allRew : Term QidList MachineInt  -> TermList .
op topRew : Term Qid MachineInt MachineInt -> TermList .
op lowerRew : Term Qid MachineInt -> TermList .
op rewArgs : Qid TermList TermList Qid MachineInt -> TermList .
op rebuild : Qid TermList TermList TermList -> TermList .

eq meta-apply'(T, L, N, M) =
    extTerm(meta-apply(MOD, T, L,
                createSubst(num-metavar, M), N)) .

eq allRew(T, nil, M) = ~ .
eq allRew(T, L LS, M) = topRew(T, L, 0, M) ,
                         lowerRew(T, L, M) , allRew(T, LS, M) .
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) = rewArgs(OP, ~, TL, L, M) .
eq rewArgs(OP, Before, T, L, M) =
    rebuild(OP, Before, allRew(T, L, M), ~) .
eq rewArgs(OP, Before, (T, After), L, M) =
    rebuild(OP, Before, allRew(T, L, M), After) .
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) .

*** Term sets
op `{`} : -> TermSet .
op _U_ : TermSet TermSet -> TermSet [assoc comm id: {}] .
op _isIn_ : Term TermSet -> Bool .
ceq T U T' = T if meta-reduce(MOD, '_==_[T, T']) == {'true}'Bool .
eq T isIn {} = false .
ceq T isIn (T' U TS) = true
     if meta-reduce(MOD, '_==_[T, T']) == {'true}'Bool .
ceq T isIn (T' U TS) = T isIn TS
     if meta-reduce(MOD, '_=/=_[T, T']) == {'true}'Bool .

*** Definition of the depth-first strategy
*** (with substitution of "metavariables")
op <_`,_> : Term MachineInt -> Pair .
op nil : -> PairSeq .
op _|_ : PairSeq PairSeq -> PairSeq [assoc id: nil] .

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) .

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 + num-metavar)) | PS )
          fi)
    fi .

*** All solutions
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 + num-metavar)) | PS)
         fi)
    fi .
endfm)

(mod CCS-OK is
  protecting CCS-SEMANTICS .
  ops p1 p2 p3 : -> Process .
  eq p1 = ('a . 0) + ('b . 0 | ('c . 0 + 'd . 0)) .
  eq p2 = ('a  . 'b . 0 | (~ 'c . 0) ['a / 'c]) \ 'a .
  eq p3 = ('Proc | ~ 'a . 'b . 0) \ 'a .
  sort Answer .
  ops solution no-solution maybe-sol : -> Answer .
  op ok : Configuration -> Answer .
  vars A A' : Act .
  var P : Process .
  var P? : Process .
  vars JS JS' : JudgementSet .
  eq ok({{ emptyJS }}) = solution .
  ceq ok({{ JS }}) = maybe-sol if JS =/= emptyJS .
  eq ok({{ emptyJS | JS' }}) = solution .
  ceq ok({{ JS | JS'}}) = maybe-sol if JS =/= emptyJS .
  eq context =
     (('Proc   =def  'a . tau . 'Proc) &
      ('Ven    =def  '2p . 'VenB  +  '1p . 'VenL)  &
      ('VenB   =def  'big . 'collectB . 'Ven)    &
      ('VenL   =def  'little . 'collectL . 'Ven)   &
      ('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 )))  .
endm)

(mod META-CCS is
  including META-LEVEL .
  op METACCS : -> Module .
  eq METACCS = up(CCS-OK) .
  op labelsIDs : -> QidList .
  eq labelsIDs = ( 'bind 'dist 'pref 'sum 'par 'res 'rel
                   'def 'nil 'seq 'refl 'tran 'act ) .
endm)

(view ModuleCCS from AMODULE to META-CCS is
  op MOD to METACCS .
```

```
  op num-metavar to 3 .
  op labels to labelsIDs .
endv)

(mod SEARCH-CCS is
  protecting SEARCH[ModuleCCS] .
endm)

(fmod CCS-SUCC is
  pr SEARCH-CCS .
  op succ : Term TermSet -> TermSet .
  op wsucc : Term TermSet -> TermSet .
  op filter : TermSet TermSet -> TermSet .
  op filter-bind : TermList TermSet -> TermSet .
  var A T T' BS : Term .
  vars MVS TS AS : TermSet .
  var TL : TermList .

  eq filter({}, MVS) = {} .
  eq filter(_U_( '`{`{_|_`}`}[{'emptyJS}'JudgementSet, BS], TS),
            MVS) = _U_(filter-bind(BS, MVS), filter(TS, MVS)) .
  eq filter-bind( ('`[_:=_`] [T, T']) , MVS) =
     if (T isIn MVS) then T' else {} fi .
  eq filter-bind( ('`[_:=_`] [T, T'], TL) , MVS) =
     if (T isIn MVS) then _U_(T', filter-bind(TL, MVS))
     else filter-bind(TL, MVS)
     fi .
  eq filter-bind('__ [ TL ] , MVS) = filter-bind(TL, MVS) .

  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) .
  eq wsucc(T, {}) = {} .
  eq wsucc(T, A U AS) = filter(allSol(
         '`{`{_|_`}`} ['_==_==>_[T, A, '?`(_`)P[{''proc}'Qid]],
         {'emptyJS}'JudgementSet]), '?`(_`)P[{''proc}'Qid])
      U wsucc(T,AS) .
endfm)

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

```
  op <_>_ : TermSet HMFormula -> HMFormula .
  op `[`[_`]`]_ : TermSet HMFormula -> HMFormula .
  op <<_>>_ : TermSet HMFormula -> HMFormula .
  op forall : TermSet HMFormula -> JudgementSet .
  op exists : TermSet HMFormula -> JudgementSet .

  sort Judgement .
  op _|=_ : Term HMFormula -> Judgement .
  sort JudgementSet .
  op emptyJS : -> JudgementSet .
  subsort Judgement < JudgementSet .
  op __ : JudgementSet JudgementSet -> JudgementSet
          [assoc id: emptyJS] .

  var P : Term .
  var 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 .
  rl [box]  : P |= [[ K ]] Phi => forall(wsucc(P, K), Phi) .
  rl [diam] : P |= << K >> Phi => exists(wsucc(P, K), Phi) .
endm)

(mod MODAL-LOGIC-OK is
  protecting MODAL-LOGIC .
  sort Answer .
  ops solution no-solution maybe-sol : -> Answer .
  op ok : JudgementSet -> Answer .
  var JS : JudgementSet .
  eq ok(emptyJS) = solution .
  ceq ok(JS) = maybe-sol if JS =/= emptyJS .
endm)

(mod META-MODAL-LOGIC-OK is
  including META-LEVEL .
  op METAMODALCCS : -> Module .
  eq METAMODALCCS = up(MODAL-LOGIC-OK) .
  op labelsIDs : -> QidList .
  eq labelsIDs = ( 'true 'and 'or 'box 'diam 'ex 'box 'diam ) .
endm)
```

```
(view ModuleMODAL-LOGIC-OK from AMODULE to META-MODAL-LOGIC-OK is
  op MOD to METAMODALCCS .
  op num-metavar to 0 .
  op labels to labelsIDs .
endv)

(mod SEARCH-MODAL-CCS is
  protecting SEARCH[ModuleMODAL-LOGIC-OK] .
endm)
```

B Examples in Maude

The examples used in this paper as they are introduced in the Maude system are as follows:

```
*** Section 2.1

(select CCS-SEMANTICS .)
(rew {{ 'a . 'b . 0 -- 'a -> 'b . 0 }} .)
(rew {{ 'a . 0 + 'b . 0 -- 'b -> 0 }} .)

*** Section 2.3

(select SEARCH-CCS .)
(red allRew(up(CCS-OK, {{ 'a . 0 + 'b . 0 -- 'b -> 0 }}),
            labelsIDs, 0) .)

*** Section 2.4

(red rewDepth(up(CCS-OK,{{ p1 -- 'c -> ('b . 0 | 0) }})) .)
(red rewDepth(up(CCS-OK,
     {{ p2 -- 'a -> ('b . 0 | (~ 'c . 0) ['a / 'c]) \ 'a }})) .)
(red rewDepth(up(CCS-OK,
     {{ p2 -- tau -> ('b . 0 | 0 ['a / 'c]) \ 'a }})) .)
(red rewDepth(up(CCS-OK,
     {{ p3 -- tau -> (tau . 'Proc | 'b . 0) \ 'a }})) .)

*** Section 3.2

(red rewDepth(up(CCS-OK, {{ p1 -- 'c -> ?('any)P }})) .)
(red rewDepth(up(CCS-OK, {{ p2 -- 'a -> ?('any)P }})) .)

*** Section 3.3

(select CCS-SUCC .)
(red succ(up(CCS-OK, p1), up(CCS-OK, 'a) U up(CCS-OK, 'b)) .)
```

```
*** Section 5.2

(red rewDepth(up(CCS-OK, {{ p2 == 'b ==> ?('any)P }})) .)
(red rewDepth(up(CCS-OK, {{ 'Proc = 'a 'a 'a ==> 'Proc }})) .)
(red wsucc(up(CCS-OK, 'Proc), up(CCS-OK, 'a)) .)

*** Section 6.3

(select SEARCH-MODAL-CCS .)
(red rewDepth(up(MODAL-LOGIC-OK,
  up(CCS-OK, 'Ven) |= [up(CCS-OK, 'big) U
                       up(CCS-OK, 'little)] ff)) .)
(red rewDepth(up(MODAL-LOGIC-OK,
  up(CCS-OK, 'Ven) |= [up(CCS-OK, '2p)](
  ([up(CCS-OK, 'little)] ff) /\ (< up(CCS-OK, 'big) > tt)))) .)
(red rewDepth(up(MODAL-LOGIC-OK,
  up(CCS-OK, 'Ven) |= [up(CCS-OK, '1p) U up(CCS-OK, '2p)]
                      [up(CCS-OK, '1p) U up(CCS-OK, '2p)] ff)) .)
(red rewDepth(up(MODAL-LOGIC-OK,
  up(CCS-OK, 'Ven) |= [up(CCS-OK, '1p) U up(CCS-OK, '2p)]
         [up(CCS-OK, 'big) U up(CCS-OK, 'little)]
         < up(CCS-OK, 'collectB) U up(CCS-OK, 'collectL) > tt)) .)
(red rewDepth(up(MODAL-LOGIC-OK,
  up(CCS-OK, 'Crossing) |= [[up(CCS-OK, 'car)]]
    [[up(CCS-OK, 'train)]] ( (<< up(CCS-OK, ~ 'ccross) >> tt) \/
                             (<< up(CCS-OK, ~ 'tcross) >> tt)))) .)
(red rewDepth(up(MODAL-LOGIC-OK,
  up(CCS-OK, 'Crossing) |= [[up(CCS-OK, 'car)]]
                           [[up(CCS-OK, 'train)]]
                           ( (<< up(CCS-OK, ~ 'ccross) >> tt) /\
                             (<< up(CCS-OK, ~ 'tcross) >> tt)))) .)
```

C Complete Isabelle code

The complete Isabelle specification is the following one.

theory *CCSSyntax = Main :*

types *id = "char list"*

datatype *label = nm id*
            | conm id

consts *compl :: "label ⇒ label"*
primrec *"compl (nm a) = conm a"*
        "compl (conm a) = nm a"

datatype *action = tau*

```
              | va label

datatype process = Stop                  ("0")
       | Prefix action  process      ("_ . _" [120, 110] 110)
       | Sum    process process      (infixl "+" 85)
       | Par    process process      (infixl "|" 90)
       | Rel    process label label ("_ [ _ =: _ ]" [100,120,120] 100)
       | Rest   process label        ("_ \ _" [100, 120] 100)
       | Id id

datatype Def = Defi id process
datatype Context = Empty
                 | And Def Context

consts defin :: "id ⇒ Context ⇒ process"
primrec
  "defin X Empty = Stop"
  "defin X (And D C') =
    (case D of (Defi Y P) ⇒ (if (X = Y) then P else (defin X C')))"

consts Context :: Context

end


theory CCSSemantics = CCSSyntax :

consts trans :: "(process * action * process) set"
syntax "_trans" :: "[process, action, process] ⇒ bool"
                    ("_/ - _ →/ _" [80,0,80] 70)

translations  "P - a → P'" == "(P,a,P') ∈ trans"

inductive trans
intros
  prefi: "A . P - A → P"
  sum1:  "P1 - A → P ⟹ P1 + P2 - A → P"
  sum2:  "P2 - A → P ⟹ P1 + P2 - A → P"
  par1:  "P1 - A → P ⟹ P1 | P2 - A → P | P2"
  par2:  "P2 - A → P ⟹ P1 | P2 - A → P1 | P"
  par3:  "⟦ P1 - va L → P ; P2 - va (compl L) → Q ⟧
       ⟹  P1 | P2 - tau → P | Q"
  rel1:  "⟦ P - va N → P' ; N ≠ L ; N ≠ compl L ⟧
       ⟹  P [ M =: L ] - va N → P' [ M =: L ]"
  rel2:  "P - tau → P' ⟹ P [ M =: L ] - tau → P' [ M =: L ]"
  rel3:  "P - va L → P' ⟹   P [ M =: L ] - va M → P' [ M =: L ]"
  rel4:  "⟦ P - va N → P' ; N = compl L ⟧
       ⟹  P [ M =: L ] - va (compl M) → P' [ M =: L ]"
```

```
  res1:  "P - tau → P' ⟹ P \ L - tau → P' \ L"
  res2:  "⟦ P - va L → P' ; L ≠ M ; compl L ≠ M ⟧
       ⟹ P \ M - va L → P' \ M"
  defi:  "(defin X Context) - A → Q ⟹ Id X - A → Q"

end



theory CCSOK = CCSSemantics :

consts p1 :: process
defs p1_def[simp]: "p1 ≡ ( va (nm ''a'') . 0) +
     (va (nm ''b'') . 0 | (va (nm ''c'') . 0 + va (nm ''d'') . 0))"

consts p2 :: process
defs p2_def[simp]: "p2 ≡ ( va (nm ''a'') . va (nm ''b'') . 0 |
     ( va (conm ''c'') . 0) [ nm ''a'' =: nm ''c''] ) \ nm ''a''"

defs Context_def[simp]: "Context ≡
     And (Defi ''Proc'' (va (nm ''a'') . tau . Id ''Proc'')) Empty"

consts p3 ::  process
defs p3_def[simp]: "p3 ≡
   (Id ''Proc'' | va (conm ''a'') . va (nm ''b'') . 0) \ nm ''a''"


lemma "p1 - va (nm ''c'') → va (nm ''b'') . 0 | 0"
by (force intro: trans.intros)

lemma "p2 - tau  → ?P"
by (force intro: trans.intros)

lemma "p3  - tau → ?P"
by (force intro: trans.intros)

lemma "¬(p2 - va (nm ''a'') → ?P)"
apply clarsimp
by (erule trans.cases, clarsimp+)+

end


theory ModalLogic = CCSSemantics :

datatype formula =  tt
                  | ff
                  | AndF formula formula ("_ ∧ _")
```

```
                  | OrF  formula formula ("_ ∨ _")
                  | Diam "action set" formula ("⟨ _ ⟩ _")
                  | Box  "action set" formula ("[ _ ] _")

consts  sat :: "(process * formula) set"
syntax "_sat" :: "[process, formula] ⇒ bool" ("_ ⊨ _" [80,80] 70)
translations "P ⊨ Phi" == "(P,Phi) ∈ sat"

inductive sat
 intros
  ttR[intro!]:   "P ⊨ tt"
  andR:  "⟦ P ⊨ Phi ; P ⊨ Psi ⟧ ⟹ P ⊨ (Phi ∧ Psi)"
  orR1:  "P ⊨ Phi ⟹  P ⊨ (Phi ∨ Psi)"
  orR2:  "P ⊨ Psi ⟹  P ⊨ (Phi ∨ Psi)"
  diamR: "⟦ P - A → Q ; A ∈ AS; Q ⊨ Phi ⟧ ⟹ P ⊨ ⟨ AS ⟩ Phi"
  boxR:  "(∀ Q A . ( (P - A → Q ∧ A ∈ AS) ⟶ Q ⊨ Phi)) ⟹ P ⊨ [ AS ] Phi"

end
theory MODALLOGICOK = ModalLogic :

defs Context_def[simp]:  "Context ≡
  And (Defi ''Ven'' (va (nm ''2p'') . Id ''VenB'' +
                     va (nm ''1p'') . Id ''VenL''))
 (And (Defi ''VenB'' (va (nm ''big'') . va (nm ''collectB'') .
                      Id ''Ven''))
 (And (Defi ''VenL'' (va (nm ''little'') . va (nm ''collectL'') .
                      Id ''Ven''))
  Empty))"

lemma "Id ''Ven'' ⊨ ⟨{va (nm ''2p'')}⟩ (⟨{va (nm ''big'')}⟩ tt)"
apply (force intro: trans.intros intro: sat.intros)
done

lemma "Id ''VenB'' ⊨ ( ⟨{va (nm ''big''), va (nm ''little'')} ⟩ tt)"
apply (force intro: trans.intros intro: sat.intros)
done

lemma "Id ''Ven'' ⊨ ( [ {va (nm ''big''),
                         va (nm ''little'')} ] ff)"
apply (rule sat.boxR, clarify)
by (erule trans.cases, simp_all,clarsimp)+

lemma "Id ''Ven'' ⊨ [{va (nm ''2p'')}]
      (([{va (nm ''little'')}] ff) ∧ (⟨{va (nm ''big'')}⟩ tt))"
apply (rule sat.boxR, clarify)
apply (erule trans.cases, simp_all,clarsimp)+
apply (rule sat.andR, rule sat.boxR, clarify)
apply (erule trans.cases, simp_all,clarsimp)+
```

```
apply (force intro: trans.intros intro: sat.intros)
apply (rule sat.andR, rule sat.boxR, clarify)
apply (erule trans.cases, simp_all,clarsimp)+
done

lemma "Id ''Ven'' ⊨ [{va (nm ''2p''),va (nm ''1p'')}]
       ( [{va (nm ''2p''),va (nm ''1p'')}] ff)"
apply (rule sat.boxR, clarify)
apply (ind_cases "Id X - A → Q", simp)
apply (ind_cases "P + Q - A → P'")
apply (ind_cases "A . P - B → Q", clarsimp)
apply (rule sat.boxR, clarify)
apply (ind_cases "Id X - A → Q", simp)
apply (ind_cases "A . P - B → Q", clarsimp)
apply (ind_cases "A . P - B → Q", clarsimp)
apply (rule sat.boxR, clarify)
apply (ind_cases "Id X - A → Q", simp)
apply (ind_cases "A . P - B → Q", clarsimp)
done

end
```