Formal Methods in System Design, 26, 113–172, 2005

Two Case Studies of Semantics Execution in Maude: CCS and LOTOS*

ALBERTO VERDEJO alberto@sip.ucm.es
NARCISO MARTÍ-OLIET narciso@sip.ucm.es
Dpto. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain

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 and executing inference systems. In order to illustrate the general ideas we consider two substantial case studies. In the first one, we represent both the semantics of Milner's CCS and a modal logic for describing local capabilities of CCS processes. Although a rewriting logic representation of the CCS semantics is already known, it cannot be directly executed in the default interpreter of Maude. Moreover, it cannot be used to answer questions such as which are the successors of a process after performing an action, which is used to define the semantics of 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 in a general, not CCS dependent way by controlling the rewriting process by means of reflection. This executable specification plus the reflective control of rewriting can be used to analyze CCS processes. The same techniques are also used to implement a symbolic semantics for LOTOS in our second case study. The good properties of Maude as a metalanguage allow us to implement a whole formal tool where LOTOS specifications without restrictions in their data types (given as ACT ONE specifications) can be executed. In summary, we present Maude as an executable semantic framework by providing easy-tool-building techniques for a language given its operational semantics.

Keywords: rewriting logic, Maude, executable semantic framework, internal strategies, CCS, Hennessy-Milner modal logic, symbolic semantics for LOTOS, ACT ONE

1. Introduction

Rewriting logic [41, 44] was introduced in [43] 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 [40] to the idea of rewriting logic as a *logical and semantic framework*. It was shown that many other logics, widely different in nature, can be represented inside rewriting logic in a natural and direct way. The general way in which such representations are achieved is by:

- Representing formulas or, more generally, proof-theoretic structures, such as sequents, as terms in an equational data type whose equations express structural axioms natural to the logic in question.

*Research supported by CICYT projects *Desarrollo Formal de Sistemas Distribuidos* (TIC97-0669-C03-01) and *Desarrollo Formal de Sistemas Basados en Agentes Móviles* (TIC2000-0701-C02-01).

- 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.

In this way an inference rule of the form

$$\frac{S_1 \ldots S_n}{S_0}$$

can be mapped into a rewrite rule of the form $S_1 \ldots S_n \rightarrow S_0$ that rewrites *multisets* of judgements S_i. This mapping is correct from an abstract point of view, but in terms of executability of the rewrite rules it is more appropriate to consider rewrite rules of the form $S_0 \rightarrow S_1 \ldots S_n$ that still rewrite multisets of judgements but go from the conclusion to the premises, so that rewriting with these rewrite rules corresponds to searching for a proof in a goal-directed way. Again this mapping is correct, and in both cases the intuitive idea is that the rewriting relation corresponds to the horizontal bar separating conclusion from premises in the typical textbook presentation of inference rules.

These techniques can be used to naturally specify a wide variety of inference systems, as detailed in [40], including sequent systems for logics and also *structural operational semantics* definitions for languages. In particular, the similarities between rewriting logic and structural operational semantics [50] were noted in [43] and further explored in [40]. As an illustrative example, the paper [40] completely develops a representation of Milner's CCS [46] in rewriting logic, extending ideas first introduced in [45]. However, this representation of CCS cannot be directly executed in the default interpreter of Maude [11, 12, 14], a high-level language and high-performance system supporting both equational and rewriting logic computation.[1] Maude should be viewed as a *metalanguage* in which the syntax and semantics of other languages, including formal specification languages, can be formally defined [10, 16].

In order to show the executability problems of the general representation of inference systems in Maude, we use the representation of the CCS semantics [46], but we note that the problems found and our solutions are not exclusive of the CCS representation. In fact, we use them in Section 6 to implement a symbolic semantics for Full LOTOS [6, 37].

Basically, the problems with this kind of representations in Maude are the existence of new variables in the righthand side of the rewrite rules and the nondeterministic application of the semantic rules. We show how these two problems can be solved in a general way by exploiting the reflective capabilities of rewriting logic and of Maude, that allow representing rewriting logic inside itself [8, 17, 18], and in particular controlling the rewriting process. Moreover, we show how the CCS semantics representation can be extended to traces of actions and to the CCS weak transition relation [46]. This executable specification plus the reflective control of rewriting can be used to analyze CCS processes and to answer 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 [32].

The main goal of this paper has a very pragmatic character: we want to show in detail how to bridge the gap from theory (rewriting logic) to practice (Maude) so that the rewriting

logic specifications of inference systems in general, and of structural operational semantics in particular, become *executable*. For this reason, we can talk about Maude as an *executable* semantic framework, in which we provide easy-tool-building techniques for languages given their operational semantics.

To show the generality of the techniques, we have developed several case studies. The most ambitious one is Full LOTOS [37], but we believe that it is better to explain the techniques with the CCS case study that already poses the main problems (nondeterminism and new variables in the righthand side of rewrite rules) instead of using a bigger and more complex language. In addition, we also describe a Hennessy-Milner logic [32] case study that is easily built on top of CCS, and shows how the same techniques are applied to different semantics.

The Full LOTOS case study extends those techniques to a bigger language and moreover does this in such a way that the ACT ONE algebraic specifications [26] used in LOTOS to define data types are *really integrated* into the operational semantics (this is the reason for talking about Full LOTOS instead of just LOTOS), something that really breaks new ground in this approach. In addition, by means of the metalanguage features supported by Maude, the Full LOTOS semantics tool is also integrated with Full Maude [24], an extension of Maude that adds modularization and parameterization mechanisms to the language: this way in the same semantic framework we have built an entire environment with parsing, pretty printing, and input/output processing of LOTOS specifications and commands for executing them, hiding to the user the underlying use of Maude.

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 already mentioned Full Maude extension [24] to take advantage of parameterization mechanisms.[2]

1.1. Syntax and semantics representation

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 that defines actions and processes. We declare `sort(s)`, `subsort(s)`, and operators `op(s)`, which have user-definable syntax.[3] 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 are used to represent CCS labels as well as process identifiers. The importation is `protecting`, meaning that the imported module semantics is not modified. The following modules are enclosed in parentheses because they are introduced into Full Maude. In the operator declarations the symbol `_` represents the places for arguments in mixfix syntax. Comments are preceded either by `***` or by `---`.

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

```
  op ~ _ : Label -> Label . *** complementary 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 AnyContext` 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 error constant `not-defined`.

```
(fmod CCS-CONTEXT is
  protecting PROCESS .
  sorts AnyProcess Context AnyContext .
  subsort Process < AnyProcess .
  subsort Context < AnyContext .
  op _=def_ : ProcessId Process -> Context [prec 40] .
  op nil : -> Context .
  op _&_ : AnyContext AnyContext -> AnyContext
          [assoc comm id: nil prec 42] .
  op _definedIn_ : ProcessId Context -> Bool .
  op def : ProcessId Context -> AnyProcess .
  op not-defined : -> AnyProcess .
  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 keeps 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).

As said above, the general idea for the implementation in rewriting logic of the operational semantics of CCS (or any other structural operational semantics) 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 [40] the first approach was followed. In this paper, and due to executability reasons, we adopt the second one because we want to be able to prove in a goal-directed way that a given transition is valid in CCS.

We represent in Maude the CCS transition judgement $P \xrightarrow{a} P'$ by the term P `--` a `->` P' of sort `Judgement`, built with the operator[4]

```
  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, allowing in this way a more abstract treatment of syntax. 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[5] of CCS we have the semantic rule

$$\frac{P \xrightarrow{a} P'}{P\backslash l \xrightarrow{a} P'\backslash l}[a \neq l \wedge a \neq \bar{l}],$$

which is translated to the following conditional 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) .
```

Note that the side condition of the semantic rule becomes the condition of the rewrite rule.

As another example, the axiom schema

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

defining the behaviour 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. Intuitively, the idea is that we start with a transition to be proved valid and work backwards using the rewriting process in a goal-directed way, maintaining the set of transitions that have to be fulfilled in order to prove the correctness of the initial transition. When this set is empty we can conclude that the initial transition is a correct CCS transition, that is, the initial transition can be rewritten to the empty set if and only if it is a valid transition in the CCS operational semantics.

1.2. Executability problems

However, we have found two problems while working with this approach in Maude. These problems are intrinsically related to the approach itself and not to the CCS operational semantics. So, the proposed solutions are not CCS dependent and can be used to obtain executable representations of other inference systems. In Section 4 we use them to implement the Hennessy-Milner modal logic, and in Section 6 we show how we have also used this approach to implement a symbolic semantics for LOTOS [6].

The first problem is that sometimes new variables which are not present in the conclusion appear in the premises. For example, in one of the semantic rules for the parallel operator we have

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

where L is a new variable in the righthand side of the rewrite rule. Rules of this kind cannot be directly used by the Maude default interpreter; they can only be used at the metalevel using a strategy to instantiate the extra variables.

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

```
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 still arises because of possible multiple matches (modulo commutativity) against the pattern `P + Q`.

In general, not all of these possibilities lead to the empty set of judgements. So, if we were to use Maude's default interpreter, it could choose the "wrong" rule to rewrite a judgement, leading to a set of judgements that cannot be rewritten to the empty one (see Section 2.1). Thus, we have to deal with the whole conceptual tree of all possible rewrites of a judgement, and search if one of the branches leads to `emptyJS`.

1.3. Outline of the paper

In Section 2, we show how the problems described above can be solved in a general way in Maude by using reflection to obtain an executable semantics in which we can prove whether a transition $P \xrightarrow{a} P'$ is possible or not.

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, to obtain

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

We also extend the CCS semantics to sequences of actions (or *traces*) and to the weak transition semantics, which does not observe τ transitions.

In Section 4 we show how the same techniques can be used to define in Maude the semantics of the Hennessy-Milner modal logic [32] for describing local capabilities of CCS processes.

In Section 5 we describe how the CCS semantics and the Hennessy-Milner modal logic can be represented and reasoned about within the theorem prover Isabelle [49], and compare this framework with ours.

We show in Section 6 how the ideas presented in detail in this paper for the CCS semantics can also be applied to build a tool for working with Full LOTOS specifications [37], based on a symbolic semantics [6]. Although this section presents some improvements regarding the semantics representation, we have not applied them to the CCS case study because we want to present the whole process of defining an executable operational semantics in Maude step by step, from the most basic to the complex results. The LOTOS semantics representation is integrated with a translation of ACT ONE [26] data types specifications into functional modules in Maude, in a tool where Full LOTOS specifications can be entered and executed.

In Section 7 we review related work on the use of logical frameworks to represent inference systems and draw some conclusions.

We would like to point out again that our main goal in this paper is to describe in a thorough manner how we can obtain *executable* representations in Maude from the operational semantics definitions of programming languages or process algebras, or inference systems for logics in general. These executable representations can be seen as *prototypes* of interpreters for languages or provers for logics, making it possible to experiment with the different languages or logics, and also with different semantics for the same language. At this time we are not interested in using Maude as a theorem prover in order to obtain an abstract model of the semantics that would allow us to do metareasoning (such as for example the theory of bisimulation for CCS), in contrast to the objectives in most of the related work cited later in Sections 5 and 7.

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 [56], and how nondeterministic rewriting is controlled by means of a *search strategy* [5, 13]. The files with all the Maude code can be found in [59]. All the code given in the following subsection is part of the system module `CCS-SEMANTICS`.

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 .
```

Note that the constructor for metavariables begins with ?; we will also use the question mark as part of the identifiers of Maude variables for representing metavariables. Do not confuse them with input actions in value-passing CCS. It is important to use as the domain of the metavariables constructor a sort that provides an infinite number of names, to be used as needed; among other possibilities we have chosen `Qid`.

We also introduce a new sort `Act?` of "possible actions," which is the union of actions and metavariables as actions (subsort declaration below):

```
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 operations to perform the corresponding substitutions

```
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 retain 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 behaviour 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*. But `NEW1` is again a new variable (of sort `Qid`) in the righthand side of the above rules. So the modified representation also has rules with new variables in the righthand side, but now they are *more localized*: 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 main idea is that when some of these rules are really applied, the term `?(NEW1)A` is substituted by a *closed* term like `?('mv1)A` that will represent a new metavariable. The strategy presented in Section 2.3 does this as well as implementing the search in the tree of possible rewrites.

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.

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:

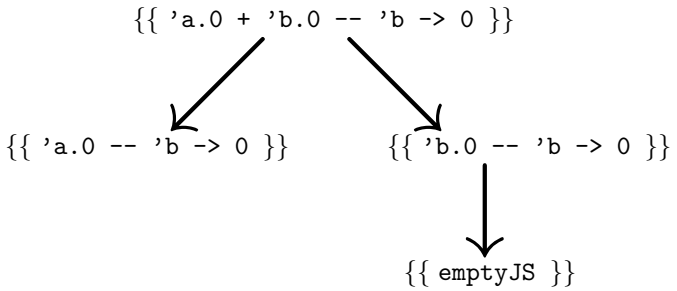
```
Maude> (rew {{ 'a.'b.0 -- 'a -> 'b.0 }} .)
result Configuration : {{ emptyJS }}
```

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 }}
```

meaning that the given transition is not possible, which is not the case. The reason 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. Before defining this search strategy, we present the Maude features that allow us to define it.

2.2. Maude's metalevel

Rewriting logic is reflective [8, 17, 18], 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 $\bar{\mathcal{R}}$ and $\bar{t}, \bar{t}'$ in $\mathcal{U}$, and we then have the following equivalence:

$$\mathcal{R} \vdash t \to t' \Leftrightarrow \mathcal{U} \vdash \langle \bar{\mathcal{R}}, \bar{t} \rangle \to \langle \bar{\mathcal{R}}, \bar{t} \rangle.$$

In Maude, the 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;
- reducing a term to normal form in a functional module and finding whether such a normal form has a given sort are reified by a metalevel function `meta-reduce`;

```
op meta-reduce : Module Term -> Term .
```

- 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 .
```

- 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 .
```

- parsing and pretty printing of a term in a signature (see Section 6.6), 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 [11]. The signature for representing terms is declared in the module `META-LEVEL` as follows:

```
sorts Term TermList .
subsorts 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 rewrites

In this section we show how the reflective properties of Maude can be used to control the rewriting of a term and the search in the tree of possible rewrites of a term. The depth-first search strategy is based on the work in [4, 5, 13], modified to deal with the substitution of metavariables explained in Section 2.1. It is completely general, so its definition is not determined in any way by the CCS semantics representation given above. It is parameterized over the module used to build rewrite trees (the module with the rewrite rules) and over the predicate that defines the search goals.

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 (which will be the `CCS-SEMANTICS` module in Section 2.4 and the `MODAL-LOGIC` module implementing the Hennessy-Milner modal logic in Section 4.3). Hence, we define a parameter *theory* to specify the requirements the parameter must satisfy: it must have a constant (`MOD`) representing the module, a constant (`labels`) representing the list of labels of rewrite rules to be applied, and a constant (`num-metavar`) representing the maximum number of metavariables that have to be provided when a rule is applied.

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

Keywords `fth...endfth` are used to define functional theories [11, 24]. Theories are used to declare module interfaces, namely the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation.

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

Since we are defining a strategy to search a tree of possible rewrites, 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), that applies to terms being rewritten and 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 any solutions below it.

The strategy will control the possible rewrites 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 in 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. If we want to use function `meta-apply` with rules with new variables in the righthand side (like `NEW1` in the rules in Section 2.1), we have to provide a substitution in such a way that the rules are always applied without new variables in the righthand side. We suppose that the new variables in the righthand side of a rule are called `NEW1, NEW2,...` These variables are then substituted by new (constant) identifiers, which are quoted numbers.[6] The operation `createSubst` receives as arguments the number of variables that have to be substituted and the greatest number already used in a substitution.

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

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 the 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(3,5)` returns the substitution

```
'NEW1@Qid <- {''6}'Qid ; 'NEW2@Qid <- {''7} 'Qid;
'NEW1@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 computing `T` and uses `createSubst` for creating a correct substitution for the new variables.

```
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 (num-metavar, M), N)) .
```

The operation `meta-apply'` returns one of the possible one-step rewrites at the top level of a given term. Next we define an operation `allRew` that returns *all* the possible *one-step sequential* rewrites [43] 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 computing `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 .  *** empty term list
var TL : TermList .
eq ~, TL = TL .
eq TL, ~ = TL .
```

Notice that this new operator for representing the empty list of terms will not be used as the identity element of the concatenation operator `_,_`, since this operator is already declared in the predefined module `META-LEVEL`. So the system will not do matching modulo identity. But we can add the previous equations, that will be used, as any other equation, from left to right to simplify terms of sort `TermList` where the empty list constant `~` might appear.

The operations needed to find all the possible rewrites, 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 possible one-step rewrites at the top of term `T` by applying rule `L`, discarding the first `N` matches, and using numbers from `M+1` on in order 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 possible one-step rewrites of the subterms of term `T` by applying rule `L`, and using numbers from `M+1` on in order to create identifiers for new variables. If `T` is a constant (a term without arguments), the empty list of terms is returned; otherwise, the operation `rewArgs` is used. This last operation is defined recursively on its third argument, which represents the arguments of `T` not yet rewritten.

```
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 list of terms `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 rewrites, and we get the metarepre-

sentation of the terms {{ 'a.0 -- 'b -> 0 }} and {{ 'b.0 -- 'b -> 0 }}.[7]

```
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 rewrites 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 checked yet 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 + num-metavar) | PS)
           fi)
     fi .
```

We have defined a search strategy at the metalevel to solve the "problem" of nondeterministic application of rewrite rules; this nondeterminism is essential to rewriting logic. The future

version of the Maude system [15] will include a `search` command to look in the tree of all rewrites of a term for terms that match a given pattern, and this will bring efficiency advantages since the search will be integrated in the system implementation. However, having defined the search at the metalevel has the advantage of allowing us to make it more general. For example, we use a user-definable `ok` predicate to define search goals. Moreover, we have been able to include the generation of new metavariables in the search process. For all these reasons we could not use directly the new search command.

2.4. Some examples

Now we can test the formalization in Maude of the CCS semantics with some examples involving different judgements. First, we specify a module `CCS-OK` that extends the CCS syntax and semantic rules by defining some process constants to be used in the examples, assigning a value to the constant `context` used by the semantics, and specifying 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, meaning 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 the module `CCS-OK`. We use the Full Maude `up` function [11, 24] 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 [11, 24] in order to instantiate the parameterized generic module SEARCH with it. Views are used to assert how a particular target module or theory is claimed to satisfy a source theory. In the definition of a view we have to indicate its name, the source theory, the target module or theory, and the mapping for each sort or operator.

```
(view ModuleCCS from AMODULE to META-CCS is
  op MOD to METACCS.
  op labels to ('bind 'pref 'sum 'par 'res 'dist 'rel 'def) .
  op num-metavar to 3 .
endv)
```

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

Module SEARCH-CCS includes the instantiation of the search strategy with the module containing the CCS semantics, constant processes, CCS context, and the predicate ok defined above.

Now we can test the examples using this module. 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 at the end of Section 3.1 how this can be proved.

The correctness of the obtained proofs is dependent on the correctness of the search strategy; if it were incomplete, for example because it does not apply a rule that can be applied, the "proof" that a process cannot perform a transition would not be valid. The positive answers could also be invalid, for example if judgements to be proved were lost by the search in some cases. However, the simplicity and generality of the search strategy are advantages regarding these problems. Once we have a correct strategy that truly builds and traverses the tree of all rewrites of a term, we have a correct strategy forever, since it does not require any change when we modify the semantics representation below it, or even if we change the semantics at all, for example, for other languages.

3. Extensions 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 a transition $P \xrightarrow{a} P'$ is possible for some unspecified P', i.e., P' is *unknown*. This is the same problem we found when new variables appeared 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:[8]

```
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 judgement 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 code in [59]).

Now we can prove that process `p1` (see Section 2.4) 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*
```

3.2. *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 rewrites will bind the metavariable `?('proc)P` with the successors of `P`, but two problems arise. The first one is that we lose the bindings between metavariables and processes when they are substituted by the rewrite rule bind (see Section 2.1). To solve this, we have to modify the operator that builds configurations by keeping the 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 new bind rule:

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

We also have to change the operation ok; now a configuration is a solution if its first part represents the empty set of judgements:

```
vars JS JS' : JudgementSet.
eq ok({{ empty JS | 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, that is, in order to explore (in a depth-first way) the whole tree of rewrites to find *all* the nodes that satisfy the predicate ok. The operation allSol, added to the module SEARCH, returns a set of 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 + num-metavar) | PS)
        fi)
fi.
```

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

process[9]

```
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` 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 the module `CCS-OK` in Section 2.4. For example, we can compute the successors of process `p1` after performing either action `'a` or action `'b`:

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

3.3. *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 now 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

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

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:

```
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 .
```

3.4. Extension to weak transition semantics

Another important transition relation defined in CCS, $P \stackrel{a}{\Rightarrow} P'$, does not observe τ transitions [46]. It is defined as

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

where $(\stackrel{\tau}{\rightarrow})^*$ denotes the reflexive, transitive closure of $\stackrel{\tau}{\rightarrow}$. It is defined in the following way for an arbitrary action a

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

Finally, the weak transition semantics is extended to traces as follows:

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

We show in this section how these extensions can be specified 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? .
```

Note that the basic transition $\xrightarrow{a}$ is included in its reflexive, transitive closure $(\xrightarrow{a})^*$, by using first the `tran` rule and then one of the `refl` rules to rewrite the second premise.

In the same way, we can define in Maude the weak transition relation for both actions and 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 compute the successors of a process with respect to the weak transition semantics:

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

```
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) .
```

Continuing with our running example, the module `CCS-OK` of 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 also prove that the recursive process `'Proc` can execute successively three observable `'a`'s and then become the same process:

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

Finally, we can compute 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
```

4. A 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 the previous sections.

4.1. 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 [32] and its semantics is presented in [57]. Formulas are as follows

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

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

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

4.2. 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. We use the same name for the sorts `Judgement` and `JudgementSet` but notice that they are different from the ones declared in the module `CCS-SUCC`.

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

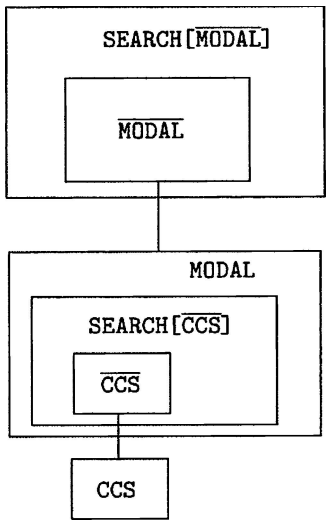


Figure 1. Module structure for the modal logic.

```
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 of the _U_ operator.

We can instantiate the parameterized module `SEARCH` in Section 2.3 with the metarepresentation of the module containing the definition of the modal logic semantics, so we obtain the hierarchy of modules shown in figure 1. At the bottom we have the module `CCS` with the CCS semantics implementation extended with the `ok` predicate needed by the search strategy. Its metarepresentation $\overline{\texttt{CCS}}$ is used to instantiate the search strategy, and this instantiation is used to define the Hennessy-Milner modal logic semantics in the module `MODAL`. In turn, this module extended with its own definition of the predicate `ok` is metarepresented, producing $\overline{\texttt{MODAL}}$, and used to instantiate again the search strategy.

4.3. Examples

As an example taken from [57], we show how to prove some modal formulas satisfied by a vending machine `'Ven` defined in a CCS context by

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

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

One of the properties that the vending machine fulfills is that a button cannot be pressed before any money is deposited. This corresponds to the formula

$$\texttt{Ven} \models \texttt{[big, little]ff}$$

which can be proved in Maude:

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

Another interesting property that Ven satisfies is that after a 2p coin is inserted the little button cannot be pressed whereas the big one can:

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

Ven also satisfies that once 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 pressed, 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
```

4.4. More modalities

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

$$P \models [\![K]\!]\ \Phi \text{ iff } \forall\ Q \in \{P' \mid P \stackrel{a}{\Longrightarrow} P' \wedge a \in K\}.\ Q \models \Phi$$
$$P \models \langle\!\langle K \rangle\!\rangle\ \Phi \text{ iff } \exists\ Q \in \{P' \mid P \stackrel{a}{\Longrightarrow} P' \wedge a \in K\}.\ Q \models \Phi$$

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` of Section 3.4 to compute the successors of process `P` with respect to the weak transition semantics (compare with the representation of the previous modalities in Section 4.2).

We can now prove some properties of a railroad crossing system [57] 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 system consists of three components: `Road`, `Rail`, and `Signal`. Actions `car` and `train` represent the approach of a car and a train, `up` opens the gates for the car, $\overline{\texttt{ccross}}$ is the car crossing, `down` closes the gates, `green` is the receipt of a green signal by the train, $\overline{\texttt{tcross}}$ is the train crossing, and `red` sets the light red.[10]

The process `Crossing` satisfies that whenever a car and a train arrive to the crossing, only one of them is allowed 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*
```

5. Comparison with Isabelle

In this section we describe how the CCS semantics and the Hennessy-Milner modal logic can be represented and reasoned about within the theorem prover Isabelle [49], and compare this framework with ours.

5.1. *A formalization of CCS in Isabelle*

Isabelle is a generic system for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, Higher-Order Logic [49].

Working with Isabelle means creating theories, which are named collections 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 HOL formulas. Terms are formed as in functional programming by applying functions to arguments, and formulas are terms of type *bool*.

Isabelle distinguishes between 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 can also define the CCS contexts (the complete code is available in [59]).

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 formalised 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 translates the rules into a standard form with all free variables converted into schematic ones, so that they can be instantiated through unification.

These are some of the rules defining the CCS semantics:

```
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 ease this process of 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** implicitly 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:)`. The 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 Isabelle formalization of 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 (nm ''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 at the end of Section 3.1.

```
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+)+
```

Note how an explicit negation is used in this case. We compare this with our approach in the following section.

We can also 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!]:  " ⊨ 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)
```

In the following we show how some properties about the vending machine of Section 4.3 can be proved in Isabelle.

```
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 ''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
```

5.2. 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, and are already built in the system, 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 by creating new fresh metavariables and propagating their values when they become known.

Isabelle/HOL has also 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.

As we commented in Section 2.4, negative proofs (for example, that a CCS transition is not valid) in our approach rely on the completeness of our search strategy. In Isabelle a theorem with an explicit negation can be proved by using induction.

Isabelle, as a logical framework, uses a higher-order logic to metarepresent the user-defined object 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 case study 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 like 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 successors of a process. 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 directly compute (or show) this set. However, Berghofer [1] has developed a mechanism for obtaining programs from Isabelle specifications by interpreting inductively defined predicates as logic programs. These Prolog-like programs (translated into functional programs) let you enumerate solutions [1, 2].

Another advantage of using Maude is that it has been designed to be used as a metalanguage, where complete tools (including parsing, pretty printing, and input/output processing) for the execution of other languages can be easily built. In Section 6.6 we will describe how this is done for LOTOS in our second case study.

Isabelle/HOL has also been used to represent CCS semantics by Christine Röckl in her PhD thesis [53]. She defines the CCS transition relation in an inductive way as we have done and then defines equivalence relations between processes, like bisimulation and observation equivalence, and proves properties about them in Isabelle. Röckl has also used Isabelle/HOL to represent the π-calculus in different ways, with first-order and higher-order abstract syntax [52–54].

HOL [29], a different but similar system to Isabelle/HOL, has been used by Monica Nesi to formalise value-passing CCS [48] and a modal logic for it [47]. In both cases she is interested in proving properties about the represented semantics themselves. Melham has also used the HOL theorem prover to present a mechanized theory of the π-calculus [42].

In Section 7 we review some related works that use other logical frameworks to represent CCS and other calculi.

6. Executing LOTOS

In this section we go one step further in the implementation of operational semantics in Maude. We present a formal tool where LOTOS specifications without restrictions in their data types can be executed. As we will see, the reflective features of rewriting logic and the metalanguage capabilities of Maude make it possible to implement the whole tool in the same semantic framework, and have allowed us to implement the LOTOS semantics and to build an entire environment with parsing, pretty printing, and input/output processing of LOTOS specifications.

The formal description technique LOTOS [37] was developed within ISO for the formal specification of open distributed systems. Its behaviour description part is based on process algebras, borrowing ideas from CCS [46] and CSP [34], and the mechanism to define and to deal with data types is based on ACT ONE [26]. The union of the behaviour and data type description parts is known as Full LOTOS. We use in this paper the term LOTOS to refer to the whole language. LOTOS became an international standard in 1989; since then, LOTOS has been used to describe hundreds of systems, and most of this success is due to the existence of tools where specifications can be executed, compared, and analyzed [20, 25, 27, 28, 30].

The standard defines LOTOS semantics by means of labelled transition systems, where each data variable is instantiated by every possible value. That is the reason why most tools ignore or restrict the use of data types. Calder and Shankland [6] have recently defined a symbolic semantics for LOTOS which gives meaning to symbolic or data param-eterised processes (see below) and avoids infinite branching.

6.1. LOTOS symbolic semantics

A symbolic semantics for LOTOS is given by associating a symbolic transition system with each LOTOS behaviour expression [6]. Following [31], Calder and Shankland define *symbolic transition systems* (STS) as transition systems which separate data from process behaviour by making the data symbolic. STS's are labelled transition systems with variables, both in states and transitions, and conditions which determine the validity of a transition. A symbolic transition system consists of:

- A (nonempty) set of states. Each state T is associated with a set of free variables, denoted $fv(T)$.
- A distinguished initial state, T_0.
- A set of transitions written as $T \xrightarrow{b\ \alpha} T'$, where α is a simple or structured event and b is a Boolean expression, such that $fv(T') \subseteq fv(T) \cup fv(\alpha), fv(b) \subseteq fv(T) \cup fv(\alpha)$, and $\#(fv(\alpha) - fv(T)) \leq 1$.

In the symbolic semantics, *open* behaviour expressions label states (for example, the behaviour $h!x$; **stop**), and transitions offer variables, under some conditions; these conditions determine the set of values which may be substituted for variables.

In [6] the intuition and key features of this semantics are presented, together with axioms and transition rules for each LOTOS operator. We present here only some of the semantic rules. These are the rules whose translation into Maude will be given in Section 6.3 as illustration of our implementation techniques. The remaining rules together with their implementation are available in [60].

$$a; P \xrightarrow{tt\ a} P \qquad g?x : S[SP]; P \xrightarrow{SP\ gx} P$$

$$\frac{P \xrightarrow{b\ \alpha} P'}{([SP] -> P) \xrightarrow{b \wedge SP\ \alpha} P'}$$

$$\frac{P \xrightarrow{b\ \alpha} P'}{hide\ g_1, \ldots, g_n\ in\ P \xrightarrow{b\ i} hide\ g_1, \ldots, g_n\ in\ P'}$$

if **name** $(\alpha) \in \{g_1, \ldots, g_n\}$

$$\frac{P \xrightarrow{b\ \alpha} P'}{hide\ g_1, \ldots, g_n\ in\ P \xrightarrow{b\ \alpha} hide\ g_1, \ldots, g_n\ in\ P'}$$

if **name** $(\alpha) \notin \{g_1, \ldots, g_n\}$

$$\frac{P_1 \xrightarrow{b\ \alpha} P_1'}{P_1|[g_1, \ldots, g_n]|P_2 \xrightarrow{b\sigma\ \alpha\sigma} P_1'\sigma[g_1, \ldots, g_n]|P_2}$$

name $(\alpha) \notin \{g_1, \ldots, g_n, \delta\}$

$$\sigma = \begin{cases} [z/x] & \text{if } \alpha = gx \text{ and } x \in \mathit{vars}(P_2) \quad \text{where } z \in \textbf{new-var} \\ [\,] & \text{otherwise} \end{cases}$$

6.2. Representing LOTOS syntax in Maude

In this section we introduce the abstract LOTOS syntax. It is defined in the Maude functional module `LOTOS-SYNTAX`. We use quoted identifiers to internally maintain LOTOS identifiers of variables, sorts, and gates. Booleans are the only predefined data type. This syntax is not complete, since LOTOS syntax may be extended in a user-definable way when ACT ONE data type specifications are used. Values of these data types will be included in the type `DataExp` below; we will see how this is done in Section 6.6. For simplicity, as it is done in [6], we will assume that only one event offer can occur at an action.

```
(fmod LOTOS-SYNTAX is protecting QID .
  sorts VarId SortId GateId .
  op V: Qid -> VarId . op S: Qid -> SortId . op G: Qid -> GateId .
  sort DataExp .
  subsort VarId Bool < DataExp .
  sort BehaviourExp .
  op stop : -> BehaviourExp .
  sorts SimpleAction StrucAction Action .
  subsort GateId < SimpleAction .
  subsorts SimpleAction StrucAction < Action .
  op __ : GateId Offer -> StrucAction [prec 30] .
  op !_ : DataExp -> Offer [prec 25] .
  op ?_ : IdDecl -> Offer [prec 25] .
  op _;_ : Action BehaviourExp -> BehaviourExp [prec 35] .
  op _`[`]_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .
  sort GateIdList .
  subsort GateId < GateIdList .
  op _`,_ : GateIdList GateIdList -> GateIdList [prec 35] .
  op _|`[_`]|_  : BehaviourExp GateldList BehaviourExp ->
        BehaviourExp [prec 40] .
  op hide_in_ : GateIdList BehaviourExp -> BehaviourExp [prec 40] .
  [. . .]
  endfm)
```

6.3. Representing the LOTOS semantics in Maude

In order to represent in Maude the LOTOS symbolic semantics we follow the ideas presented in Section 2. Here we describe only the main or new ideas.

First, we define the elements of a symbolic transition, that is, events and transition conditions.

```
(mod LOTOS-SYMBOLIC-SEMANTICS is
  protecting LOTOS-SYNTAX .
  sorts SimpleEv StructEv Event TransCond .
  subsort SimpleAction < SimpleEv .
  op delta : -> GateId .
  op __ : GateId DataExp -> StructEv .
  subsort Bool SelecPred < TransCond .
  op _=_ : DataExp DataExp -> TransCond [prec 25] .
```

We define a sort of `Judgements` to represent the basic elements of a semantic rule. Common judgements will be LOTOS transitions, but we will see below other kinds of judgements.

In Section 2, we worked with sets of judgements. In this case, for efficiency reasons, more concretely, in order to avoid multiple matching modulo commutativity, we work with sequences of judgements. This means that judgements will be ordered and they will be proved from left to right, that is, if the first judgement of a sequence cannot be reduced to the empty sequence, then we know that the whole sequence cannot be reduced and we can abandon it.

```
  sorts Judgement JudgSeq .
  op emptyJS : -> JudgSeq .
  subsort Judgement < JudgSeq .
  op __ : JudgSeq JudgSeq -> JudgSeq [assoc id: emptyJS prec 60] .
  var J : Judgement. vars JS JS' : JudgSeq .
  sort Configuration .
  op `{`{_|_`}`} : JudgSeq JudgSeq -> Configuration .
```

However, this change may affect the way premises are written in a semantic rule, because the reduction of a judgement should not require bindings produced by a later (on the right) judgement.

In order to solve the problem of new variables in the righthand side of a rewrite rule, we also use explicit metavariables. Metavariables are needed for transition conditions, events, and behaviour expressions.

In the case of metavariables as transition conditions, we define a new sort `MVTransCond`, an operator for building new metavariables from quoted identifiers, and also a new sort `TransCond?` of "possible transition conditions."

```
sorts MVTransCond TransCond? .
subsorts MVTransCond TransCond < TransCond? .
op ?‘(_‘)b : Qid -> MVTransCond .
```

We also need a judgement for representing the binding of a metavariable to a concrete value, and rules to propagate this binding to the rest of judgements. In Section 2.1 we used auxiliary operations (like `<act_:=_>_`) to perform the substitution of values for metavariables. We defined them by means of equations that distinguish cases based on the constructors of the terms where the substitution is being applied. We cannot do the same now, because we do not know the syntax of data expressions. Instead, we use the operator `_[[_/_]]` to represent *a syntactic substitution*. The improvement is that we can define the behaviour of this operator at the metalevel only *once*, as we will see in Section 6.5, instead of using several overloaded auxiliary operations as we have done in Sections 2.1 and 3.1. A judgement for representing the equality of transition conditions is also needed. It is eliminated when both metavariables have been bound to the same concrete value.

```
op ‘[_:=_‘] : MVTransCond TransCond? -> Judgement .
op _‘[‘[_/_‘]‘] : JudgSeq TransCond MVTransCond -> JudgSeq .
vars b b’ : TransCond .
var ?b : MVTransCond .
var b? : TransCond? .
rl [bind] : {{ [?b := b] JS | JS }} =>
{{ (JS[[ b / ?b]]) | [?b := b] JS’ }}
op ‘[_==_‘] : TransCond? TransCond? -> Judgement .
rl [equal] : [b == b] => emptyJS .
```

We do the same for metavariables as events, and metavariables as behaviour expressions (although we omit the code here).

Now we can define an operator for building symbolic transitions.

```
sort Transition . subsort Transition < Judgement .
op _--_--_-->_ : BehaviourExp TransCond? Event? BehExp? ->
                 Transition [prec 50] .
```

Before giving the semantic rules, we define several auxiliary operations used by the semantics. For example, in the semantics, a set **new-var** of fresh variable names is assumed. For building new variable names, we use the same idea as in the CCS case for building new metavariables. In the semantic rules we will use `new-var(NEW1)`, where `NEW1` is a new variable in the righthand side of the rewrite rule, and which is substituted by new quoted identifiers when the rules are applied (at the metalevel by the search strategy).

```
op new-var : Qid -> VarId .
eq new-var(Q) = V(conc(’z@, Q)) .
```

In the semantics definition a function *vars* is used to obtain the variables occurring in a behaviour expression. Since a behaviour may have data expressions, and these are built by means of a user-definable syntax, we cannot define at this level an operation to extract the variables in a behaviour. We declare an operation `vars` which is defined by means of another operation `vars@metalevel` which will be defined at the metalevel (see Section 6.5), where the behaviour (including data expressions) will be metarepresented as a `Term` and we will be able to traverse it extracting (metarepresented) LOTOS variables.

```
sorts VarSet. subsort VarId < VarSet .
op vars : BehExp? -> VarSet .
op vars@metalevel : BehaviourExp -> VarSet .
eq vars(P) = vars@metalevel(P) .
```

Other kinds of auxiliary judgements are also used in the semantics representation: a judgement for representing the fact that two actions have to be different, and a judgement enclosing a Boolean predicate (which may have metavariables). Rules `dist` and `bool` show how these judgements dissapear when they are fulfilled.

```
op `[_=/=_`] : Event? Event? -> Judgement .
op <_> : Bool -> Judgement .
crl [dist] : [a =/= a'] => emptyJS if a =/= a' .
rl [bool] : < true > => emptyJS .
```

Now, we can represent the LOTOS symbolic semantics rules in Maude. We only present some of the rewrite rules, namely those corresponding to the semantic rules shown in Section 6.1; the whole specification can be found in [60]. The first two rules correspond to the axioms for the prefix operator, and their premises (below the horizontal line) bind the metavariables in the conclusion (above the line). The third rule corresponds to the guarding operator and it is quite direct. The fourth rule corresponds to the two semantic rules for the hiding operator we saw above, and it uses an auxiliary operation hide which is reduced when the value of the metavariable `? (NEW1)a` becomes known. The fifth rule is more complicated: it deals with one of the cases of the parallel operator. Its first premise looks for possible transitions of the first component of the operator; the second premise requires that the name of the event is not in the synchronization list and it is not δ; and the rest of premises bind the metavariables. The auxiliary operation `subsPar` returns the required substitution (as shown in the corresponding rule in Section 6.1). Observe the use of the operations `vars` and `new-var`.

```
rl [sym] :        A; P -- ?b -- ?a --> ?P
         => ---------------------------------
              [?b := true] [?a := A] [?P := P] .

rl [sym] :   g ? x : s [SP] ; p -- ?b -- ?a --> ?P
         => ---------------------------------------
               [?b := SP] [?a := g x ] [?P := P] .
```

```
rl [sym] :              [SP] -> P -- ?b -- ?a --> ?P
         => ------------------------------------------------------
               P -- ?(NEW1)b -- ?a --> ?P [?b := ?(NEW1)b /\ SP] .

rl [sym] :     hide GIL in P -- ?b -- ?a --> ?P
         => ------------------------------------
               P -- ?b -- ?(NEW1)a --> ?(NEW1)P
               [?a := hide(?(NEW1)a, GIL)]
               [?a := hide GIL in ?(NEW1)P] .


op hide : Event? GateIdList -> Event? .
ceq hide(a, GIL) = i if name(a) in GIL .
ceq hide(a, GIL) = a if not(name(a) in GIL) .

rl [sym] :   P1 |[GIL]| P2 -- ?b -- ?a --> ?P
         => ------------------------------------
           P1 -- ?(NEW1)b -- ?(NEW1)a --> ?(NEW1)P
         < not(name(?(NEW1)a) in (GIL, delta) ) >
        [?b := ?(NEW1)b subsPar(?(NEW1)a, vars(P2), new-var(NEW1))]
        [?a := ?(NEW1)a subsPar(?(NEW1)a, vars(P2), new-var(NEW1))]
        [?P := (?NEW1)P subsPar(?(NEW1)a, vars(P2), new-var(NEW1))]
                |[GIL]|
           P2 .
```

Notice how we have presented the rules assuming metavariables everywhere, that is, as transition conditions, events and resulting behaviour expressions, instead of what we did for CCS where we wrote several rules depending on the places where metavariables appeared. Then, we have additional rules that reduce other kinds of transitions (without metavariables everywhere) to the above ones. For example, if we did this simplification in the CCS case, for the prefix operator we would have only the rule

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

But we would have the following *general* rules that allow other kinds of judgements as "conclusions":

```
rl [meta] :   P -- A -> P'
         => ------------------------------
            [P -- ?(NEW1)a -> ?(NEW2)P]
            [?(NEW1)a == A] [?(NEW2)P == P']
```

```
  rl [meta] :   P -- ?A -> P'
          => --------------------
             [P -- ?A -> ?(NEW2)P]
             [?(NEW2)P == P'] .

  rl [meta] :   P -- A -> ?P
          => ----------------------
             [P -- ? (NEW1)a -> ?P]
             [?(NEW1)a == A]
```

The bindings of the form `x = E`, that may appear in a transition condition when two behaviours synchronize, are not propagated to the resulting behaviour by the symbolic semantics. Actually, the value of any variable can be figured out by tracing through the conditions, traversing the symbolic transition system. However, the LOTOS tool we define below (and which uses the semantics previously defined) will propagate these bindings in order to show in a more readable way the possible transitions of a behaviour. We can define the propagation of the bindings in a transition condition at this level.

```
  op apply-subst : TransCond BehaviourExp -> BehaviourExp .
  eq apply-subst(B, P) = P .
  eq apply-subst(E1 = E2, P) =
  if (E1 : VarId) and not(E2 : VarId) then P [[ E2 / E1 ]]
  else if (E2 : VarId) and not(E1 : VarId) then
         P [[ E1 / E2 ]]
       else P fi fi .
  eq apply-subst(b /\ b', P ) = apply-subst(b, apply-subst(b',P)) .
  [...]
endm)
```

6.4. Search strategy

In order to execute the semantics we could use the general search strategy developed in Section 2.3. However, we have modified that strategy for efficiency reasons. When we are trying to rewrite a *sequence* of judgements, we only try to rewrite its *first* judgement. As we said above, the judgements are ordered, and we have to test that all of them can be rewritten to the empty sequence. So if the first judgement cannot be rewritten we do not need to rewrite the rest, because we know that in the tree of rewrites there is no node representing the empty sequence below the current node; thus, we can drop the sequence and search in other places of the tree.

Now the search strategy has the following parameters: a constant `(MOD)` representing the module, a constant `(labels)` representing the list of labels of rewrite rules to be applied, a constant `(num-metavar)` representing the number of different metavariables that can appear in the same rule, and a constant `(operators)` representing the list of operators whose arguments have to be rewritten when looking for all the rewrites of a term.

```
(fmod PARAMS is
  including META-LEVEL .
  op MOD : -> Module .
  ops labels operators : -> QidList .
  op num-metavar : -> MachineInt .
endfm)
```

This module is included by the SEARCH module, and the constants will be defined by the module which uses the search strategy, for example, the module LOTOS-TOOLS below.

6.5. Using the semantics and search strategy

Now we can instantiate the search strategy with the LOTOS semantic rules in order to make it executable. First, we specify a module LOTOS-OK extending the LOTOS syntax and semantic rules by defining the predicate ok that states when a configuration is a solution. A configuration denotes a solution when it represents the empty sequence of judgements, thus denoting that the sequence of judgements at the beginning is provable by means of the semantic rules. We define a sort PossibleTrans of possible transitions (from an initial behaviour expression). These operators are used by the operations below to represent at the object level the solutions returned by the search strategy.

```
(mod LOTOS-OK is
  protecting LOTOS-SYMBOLIC-SEMANTICS .
  sort Answer .
  ops solution no-solution maybe-sol : -> Answer .
  op ok : Configuration -> Answer .
  vars JS JS' : JudgSeq .
  eq ok({{ emptyJS | JS' }}) = solution .
  ceq ok({{ JS | JS'}}) = maybe-sol if JS =/= emptyJS .
  sort PossibleTrans .
  op |--_--_-->_: TransCond Event BehaviourExp -> PossibleTrans .
  op nil : -> PossibleTrans .
  op _&_ : PossibleTrans PossibleTrans -> PossibleTrans
           [assoc id: nil] .
endm)
```

The following module instantiates part of the constants in the module PARAMS; MOD is the only one left undefined. MOD should have the metarepresented module with the syntax and semantics of LOTOS, but, at this point, the syntax is not complete because the syntax for data values has not been defined yet: it will be later defined by the user in ACT ONE [26]. At that moment, we will be able to build a module that includes the functional modules which are the translation of the ACT ONE modules and the LOTOS syntax and semantics. We will metarepresent it and build a module with an equation that identifies the constant MOD with this metarepresented module (see Section 6.6.4).

```
(fmod LOTOS-TOOLS is
  including SEARCH .
  eq labels = ( 'bind 'dist 'equal 'bool 'sym) .
  eq num-metavar = 2 .
  eq operators = ('`{`{_|_`}`} '__) .
```

We have left two things pending, waiting for their completion at the metalevel: syntactic substitution and extraction of variables from a behaviour expression. The reason why we cannot specify them when defining the semantics is the same in both cases: the presence of data expressions with user-definable syntax. At the metalevel a fixed, known syntax is used to metarepresent terms, so we are now able to define both operations.

```
  op replace : Term Term Term -> Term .
  op replaceList : TermList Term Term -> TermList .
  ceq '_`[`[_/_`]`] [ T, Y, X ] = replace(T, Y, X) if not(T : Qid) .
  eq replace(T, Y, T) = Y .
  ceq replace({C}S, Y, X) = {C}S if X =/= {C}S .
  ceq replace(OP[TL], Y, X) =
     meta-reduce(MOD, OP[replaceList (TL, Y, X) ]) if X =/= OP[TL] .
  eq replaceList(T, Y, X) = replace(T, Y, X) .
  eq replaceList((T,TL), Y, X) = replace(T,Y,X), replaceList(TL,Y,X) .

  op vars@metalevel2 : TermList -> Term .
  ceq 'vars@metalevel [ T ] = vars@metalevel2(T) if not(T : Qid) .
  eq vars@metalevel2({C}S) = {'mt}'VarSet .
  eq vars@metalevel2(V) = {'mt}'VarSet .
  eq vars@metalevel2( 'V[TL] ) = 'V[TL] .
  ceq vars@metalevel2( F[TL] ) = vars@metalevel2(TL) if F =/= 'V .
  eq vars@metalevel2((TL,TL')) = '_U_ [ vars@metalevel2(TL),
                                        vars@metalevel2(TL') ] .
```

The operation transitions receives the metarepresentation of a LOTOS behaviour expression and returns a sequence with the metarepresentations of all its possible transitions as metarepresented terms of sort `PossibleTrans`, defined in the module `LOTOS-OK`. The transitions of process `P` are calculated by `allSol`, which searches in the tree whose root is the transition

```
    P -- ?('b)b -- ?('a)a --> ?('P)P
```

```
  op transitions : Term -> PairSeq .
  op transitions : Term MachineInt -> PairSeq .
  op get-transitions : PairSeq -> PairSeq .
  op apply-subst : PairSeq -> PairSeq .
  eq transitions(T) = transitions(T, 0) .
  eq transitions(T, N) = apply-subst(get-transitions(allSol(
```

```
        ''{{_|_'}'} ['_-- _-->_[ T,
                          '?'(_')b [ {''b}'Qid ],
                          '?'(_')a [ {''a}'Qid ],
                          '?'(_')P [ {''P}'Qid ]],
                  'emptyJS'JudgSeq ], N))) .
  [...]
endfm)
```

The operation `get-transitions` receives the solutions found and it returns a list of possible transitions, one for each solution. It filters the bindings between metavariables and values that go with each solution, looking for the concrete metavariables `?('b)b`, `?('a)a`, and `?('P)P`. The operation `apply-subst` modifies each possible transition propagating the bindings in the transition condition (if any) to the resulting behaviour expression.

The constant `MOD` should be defined by a module like the following one. It will be introduced to the Full Maude database of modules when a LOTOS specification is entered to our tool, although instead of `LOTOS-OK`, the metarepresented module will be called `EXT-LOTOS-OK` and will include the extended syntax of user-definable data types. The `up` function [24] is used to obtain the metarepresentation of the module `LOTOS-OK`.

```
(fmod FULL-LOTOS is
  including LOTOS-TOOLS .
  eq MOD = up(LOTOS-OK) .
endfm)
```

We can already use the `FULL-LOTOS` module, since it is executable. The `transitions` operation allows us to know the possible transitions of a LOTOS behaviour (although without data expressions); but it is quite cumbersome because we have to use metarepresented terms to interact with this operation.

If we want to build a usable formal tool we need more. We have to build an *environment* for it, including not only the execution aspect just described, but parsing, pretty printing, and input/output. We show how it can be done by using the metalanguage features of Maude in the next section.

6.6. Building the LOTOS tool environment

Maude has the following metalanguage features for parsing, pretty printing, and input/output [11]:

- The *syntax definition* for the language $\mathcal{L}$ is accomplished by defining a data type $\texttt{Grammar}_{\mathcal{L}}$, which can be done with very flexible user-definable *mixfix* syntax, that can mirror the concrete syntax of $\mathcal{L}$. Particularities at the lexical level of $\mathcal{L}$ can be accommodated by user-definable *bubble sorts*, that tailor the adequate notions of token and identifier to the language in question. Bubbles correspond to pieces of a module in a

language that can only be parsed once the grammar introduced by the signature of the module is available. This is specially important when $\mathcal{L}$ has user-definable syntax, as it is our case with ACT ONE.

- Parsing and pretty printing for $\mathcal{L}$ is accomplished by the functions `meta-parse` and `meta-pretty-print` in `META-LEVEL`. `meta-parse` receives as arguments the representation of a module M and the representation of a list of tokens, and returns the metarepresentation of the parsed term (a parse tree that may have bubbles) corresponding to the list of tokens for the signature of M. `meta-pretty-print` receives the representation of a module M and a term t, and returns a list of quoted identifiers that encode the string of tokens produced by pretty printing t in the syntax given by M.
- Input/output of $\mathcal{L}$ specifications and of commands for execution in $\mathcal{L}$ is accomplished by the predefined module `LOOP-MODE`, that provides a generic read-eval-print loop. This module has an operator `[_,_,_]` that can be seen as a persistent object with an input and output channel (the first and third arguments, respectively), and a state (given by its second argument). In Section 6.6.4 we will see how we can extend the state used by Full Maude. When something is written at Maude's prompt enclosed in parentheses,[11] it is placed in the first slot of the loop object as a list of quoted identifiers. The output is handled in the reverse way, that is, the list of quoted identifiers placed in the third slot of the loop is printed on the terminal.

All these techniques have been used to extend Maude itself, in the implementation of Full Maude [24]. We use this implementation not only because we want to use the auxiliary functions defined there, as we will see in Section 6.6.2, but because we need the database of modules maintained by Full Maude. Full Maude maintains as the state of the loop object a database of modules entered into the system. It is needed for the execution of commands in the modules and to perform the module operations defined by Full Maude. And we need the database to be able to build new modules *on the fly*, when LOTOS specifications are entered into the system, as we will see in Section 6.6.4.

6.6.1. The grammar of the LOTOS tool interface. We have to define first the signature (syntax) of Full LOTOS (ACT ONE and LOTOS) and the signature of the commands we are going to use in our tool to work with the entered specification.[12] We do not include here the signatures of ACT ONE and LOTOS; see [60] for the complete code.

Part of the syntax of Full LOTOS, due to the ACT ONE data types, is user-definable. As we have already mentioned, Maude provides great flexibility to define this syntax thanks to its mixfix front-end and to the use of *bubbles* [24].

The following module, `LOTOS-TOOL-SIGN`, includes the ACT ONE and LOTOS signatures and the commands of our tool.

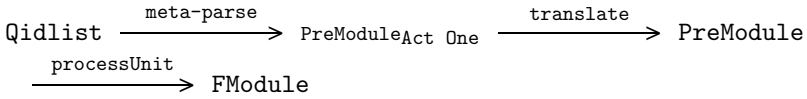
```
fmod LOTOS-TOOL-SIGN is
  protecting ACTONE-SIGN .
  protecting LOTOS-SIGN .
  sort LotosCommand .
  op show process . : -> LotosCommand .
```

```
  op show transitions . : -> LotosCommand .
  op show transitions of_. : BehaviourExp -> LotosCommand .
  op cont_ . : Machinelnt -> LotosCommand .
  op cont . : -> LotosCommand .
endfm
```

The first command is used to show the current process, that is, the behaviour expression used if we omit it in the rest of commands. The second and third commands are used to show the possible transitions (defined by the symbolic semantics) of the current or explicitly given process, that is, they start the execution of a process. The fourth command is used to continue the execution with one of the possible transitions, the one indicated in the argument of the command. `cont` is a shorthand for `cont 1`.

In order to parse some input using the built-in function `meta-parse`, we need to give the metarepresentation of the signature in which the input is going to be parsed. By including the module `LOTOS-TOOL-SIGN` in the metarepresented module `LOTOS-GRAMMAR` (not shown here) we get the metarepresentation of the signature. The `LOTOS-GRAMMAR` module will be used in calls to the `meta-parse` function in order to get the input parsed in this signature. Notice that from the call to `meta-parse` we will get a term representing the parse tree of the input (maybe with bubbles). This term will then be transformed into a term of an appropriate data type.

6.6.2. The translation of ACT ONE modules. Instead of defining a datatype for representing ACT ONE modules in Maude and operations to transform the parse tree returned by `meta-parse` into a value of this datatype, we are going to use Maude functional modules to represent (internally) ACT ONE modules. Since Full Maude already has a function (`processUnit`) to transform a parse tree (maybe with bubbles) representing a functional module into a functional module, we have to define only a function `translate` that `translates` a parse tree representing an ACT ONE module into a parse tree representing a functional module.



The operation `translate` has three arguments: the name of the LOTOS specification entered, the parse tree returned by `meta-parse` representing a list of ACT ONE datatype specifications, and a Full Maude database of modules. It returns this database modified by introducing a functional module for each ACT ONE datatype and one more module (with the name of the specification) which includes the introduced modules. This last module will be used to represent all the data types. We only show the translation of sort declarations.

```
fmod ACTONE-TRANSLATION is
  protecting DATABASE-HANDLING .
  op translate : Term Term Database -> Database .
  op translateDeclList : Term -> Term .
```

```
specification SPEC
  type Naturals is
    sorts
      nat
    opns
      0 : -> nat
      s : nat -> nat
  endtype
  type Extended-Naturals is Naturals
    opns
      _+_ : nat, nat -> nat
    eqns
      forall x, y : nat
      ofsort nat
        0 + x = x ;
        s(x) + y = s(x + y) ;
  endtype
endspec
```

```
fmod Naturals is
  including LOTOS-SYNTAX .
  including BOOL .
  sorts nat .
  subsort VarId < nat .
  subsort nat < DataExp .
  op 0 : -> nat .
  op s : nat -> nat .
endfm
fmod Extended-Naturals is
  including LOTOS-SYNTAX .
  including Naturals .
  including BOOL .
  op _+_ : nat nat -> nat .
  var x : nat .
  var y : nat .
  eq s(x) + y = s(x + y) .
  eq 0 + x = x .
endfm
fmod SPEC is
  including Naturals .
  including BOOL .
  including Extended-Naturals
endfm
```

Figure 2. An ACT ONE specification and its translation.

When an ACT ONE sort declaration for sort `T` is found, it is not only translated into a Maude sort declaration for sort `T`, but we also have to declare `T` as a subsort of sort `DataExp` (since values of the declared type could be used in a behaviour expression to be communicated) and the sort of LOTOS variables `VarId` has to be declared as a subsort of type `T` (since LOTOS variables could be used to build values of this type).

```
eq translateDeclList('sorts_['token[T]]) =
  '__['sort_ .['sortToken[T]],
     '__['subsort_ .['<_['sortToken[T], 'sortToken[{''DataExp}'Qid]]]
         'subsort_.['_<_['sortToken[{''VarId}'Qid], 'sortToken[T]]]]] .
```

Figure 2 shows an example of how an ACT ONE specification (on the left) is translated into functional Maude modules (on the right).

6.6.3. LOTOS input processing. When LOTOS behaviour expressions are introduced, either as part of a whole specification or in a tool command, they have to be transformed into elements of the data type `BehaviourExp` in the module `LOTOS-SYNTAX` (Section 6.2). The parse tree returned by `meta-parse` with module `LOTOS-GRAMMAR` may have bubbles (where data expressions may appear) that have to be parsed again using the user-defined

syntax. This syntax can be found in the functional module that includes all the modules which are the translations of the types defined in ACT ONE (see module SPEC in figure 2). Moreover, the behaviour itself can define new syntax, since it can declare new LOTOS variables by means of ? offers and these variables may appear in expressions. For example, when processing the behaviour expression

```
g ? x : nat ; h ! s(x) + s(0) ; stop
```

the data expression `s(x) + s(0)` should be parsed using the signature in the module `SPEC` extended with variable `x` of sort `nat`.

```
fmod LOTOS-PARSING is
  protecting META-LEVEL .
  protecting UNIT-DECL-PARSING .
  op parseProcess : Term Module VarSet -> Term .
  op parseAction : Term Module VarSet -> TermVars .
  [...]
endfm
```

We use the operation `parseProcess` to make this translation. It receives as arguments the term returned by `meta-parse` (representing a behaviour expression), the metarepresented module with the data types syntax (module `SPEC` in figure 2), and the set of free variables that may appear in the behaviour expression, and returns a behaviour expression without bubbles. It uses the operation `parseAction` that returns, in addition to the term metarepresenting the given action (without bubbles), the variables declared in the action (if any).

6.6.4. Extending the database by inheritance. In [24] it is explained how the persistent state of the Full Maude system is given by a single object (of class `DatabaseClass`), which maintains the database of the system. We can extend the Full Maude system by defining subclasses of `DatabaseClass` inheriting its behaviour and adding new functionalities to it, new attributes, etc.

```
mod LOTOS-DATABASE-HANDLING is
  inc EXT-DATABASE-HANDLING. pr META-LOTOS-TOOL-SIGN .
  pr ACTONE-TRANSLATION. pr LOTOS-PARSING .
  sort Lotos-DB .
  subsort Lotos-DB < DatabaseClass .
  op Lotos-DB : -> Lotos-DB .
```

We declare attributes to keep the LOTOS process we are working with, the set of possible transitions of this process (computed whenever the corresponding command is executed), and both the trace of events and the conjunction of transition conditions of transitions already executed.

```
op lotosProcess :_ : Term -> Attribute .
op transitions :_ : Term -> Attribute .
op trace :_ : Term -> Attribute .
op condition :_ : Term -> Attribute .
```

Then, we define rules that describe the behaviour associated with the new commands. We only show here—and refer to [60] for the complete code—the rule that deals with the introduction of LOTOS specifications. It says that if a specification is in the `input` attribute, then the database of modules has to be modified by introducing modules corresponding to the data types (by means of the operation `translate`), a module `EXT-LOTOS-SYNTAX` that includes the data types and the LOTOS syntax, a module `EXT-LOTOS-OK` that includes the data types and the LOTOS symbolic semantics, and a module `FULL-LOTOS` that includes the instantiation of the search strategy and defines the constant `MOD` to be equal to the metarepresentation of module `EXT-LOTOS-OK`. The `lotosProcess` attribute is also set to the process introduced in the specification (after being parsed with `parseProcess`).

```
rl [spec] :
   < O : X@ Lotos-DB |
      input : ('specification__behaviour_endspec[T,T', T'']),
      output : nil,
      db : DB, default : MN, lotosProcess : T''', Atts >
   => < O : X@Lotos-DB | input : nilTermList,
         output : ('Introduced 'specification parseModName(T)),
         db : processUnit(meta-parse(EXT-GRAMMAR,
                'fmod 'FULL-LOTOS 'is
                   'including 'LOTOS-TOOLS '.
                   'eq 'MOD '= 'up ''( 'EXT-LOTOS-OK ''') ' .
                'endfm),
              evalUnit(
                mod 'EXT-LOTOS-OK is
                  nilParameterList
                  including parseModName(T) .
                  including 'LOTOS-OK .
                  sorts none .
                  none none none none none none endm,
              evalUnit(
                fmod 'EXT-LOTOS-SYNTAX is
                  nilParameterList
                  including parseModName(T) .
                  including 'LOTOS-SYNTAX .
                  sorts none .
                  none none none none none endfm,
              translate(T, T', DB)))),
         default : parseModName(T),
```

```
         lotosProcess : parseProcess(T'', getFlatUnit(parseModName
         (T),
                          translate(T, T', DB)),mt), Atts > .
 [...]
endm
```

6.6.5. The Full Maude environment of the LOTOS tool. Finally, we give the rules to initialize the loop and to specify the communication between the loop (the input/output of the system) and the persistent state of the system. The following module is a redefinition of module FULL-MAUDE presented in [24], and it can handle both Full Maude and LOTOS commands and specifications.

```
mod LOTOS-TOOL&FULL-MAUDE is
  protecting LOTOS-DATABASE-HANDLING .
  including LOOP-MODE .
```

The init rule initializes the persistent object as an object of class Lotos-DB by initializing its attributes.

```
rl [init] : init
  => [nil,
      < o : Lotos-DB |
         db : evalUnit(CONFIGURATION,
               evalUnit(TRIV, evalUnit(UP, emptyDatabase))),
         input : nilTermList, output : nil,
         default : 'META-LEVEL,
         lotosProcess : error*,
         transitions : error*,
         condition : ({'true}'TransCond),
         trace : ({'nil}'Trace) >,
      nil].
```

There are two in rules; the first one uses the EXT-GRAMMAR to parse Full Maude modules and commands, while the second one uses the LOTOS-GRAMMAR to parse LOTOS specifications and commands; we only show the second one.

```
rl [in] :
   [QIL,
    < O : X@Lotos-DB | input : nilTermList,
                       output : nil, Atts >,
    QIL']
    => if meta-parse(LOTOS-GRAMMAR, QIL) == error*
       then [nil,
                < O : X@Lotos-DB |
```

```
                input : nilTermList,
                output : ('ERROR: 'incorrect 'input ' .), Atts >,
            QIL']
       else [nil,
                < O : X@Lotos-DB |
                input : meta-parse(LOTOS-GRAMMAR, QIL),
                output : nil, Atts >,
            QIL']
       fi .
```

The out rule, dealing with output to the Maude system, is left unmodified.

```
   crl [out] :
       [QIL, < O : X@Database | output : QIL', Atts >, QIL'']
       => [QIL, < O : X@Database | output : nil, Atts >, (QIL' QIL'')]
          if QIL' =/= nil .
endm
```

After introducing this last module into the Maude system, the extended version of Full Maude is ready to receive (Full) Maude modules. We can then introduce the modules defined in Section 6.3 in the database, so as to be able to use them when needed. At that moment the LOTOS tool becomes usable, and LOTOS specifications can be entered and executed.

6.7. Execution example

This is an example of an interaction with the LOTOS tool.

```
Maude> (specification SPEC
  type Naturals is
    [as shown above]
  endtype
  type Extended-Naturals is Naturals
    [as shown above]
  endtype
  behaviour
     h ! 0 ; stop
   []
   ( g ! (s(0)) ; stop
    |[ g ]|
      g ? x : nat ; h ! (x + s(0)) ; stop )
endspec)
Introduced specification SPEC
Maude> (show transitions .)
TRANSITIONS:
```

```
1. |-- x = s(0) -- g s(0) --> stop |[ g ]| h ! s(s(0)) ; stop
2. |-- true -- h 0 --> stop

Maude> (cont 1 .)
Trace: g s(0)                    Condition: x = s(0)
TRANSITIONS:
1. |-- true -- h s(s(0)) --> stop |[ g ]| stop

Maude> (cont.)
Trace: (g s(0)) (h s(s(0)))      Condition: x = s(0)
No more transitions .
```

7. Conclusion

We have shown how inference systems can be represented in rewriting logic and its Maude implementation in a general and *fully executable* way. In order to present the general ideas in concrete case studies, we have represented the CCS structural operational semantics and an environment for executing Full LOTOS specifications. We have solved 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 capability of the metalevel operation `meta-apply`) to solve the issue of the new variables, and a search strategy which deals with conceptual trees of rewrites to solve the problem of nondeterminism. The presented solutions are not CCS (or LOTOS) dependent and can be used to implement a wide variety of inference systems and operational semantics definitions.

The proposed methodology in the general case can be summarized as follows:

- An algebraic data type specification is used to represent the syntax of the inference system in question, including syntax for judgements and for multisets of judgements.
- An inference rule $\frac{S_1 \ldots S_n}{S_0}$ is mapped into a rewrite rule $S_0 \rightarrow S_1 \ldots S_n$ that rewrites multisets of judgements.
- New variables appearing in the righthand side of such a rule are replaced by metavariables, handled and instantiated at the metalevel. This can also be done in other places, as shown in the CCS extensions, and it is even possible to have "metavariables everywhere" as we have done in the Full LOTOS case study.
- The metavariable handling requires adding new judgements for keeping the variable instantiation and for propagating substitutions in multisets of judgements. This can be done in a systematic way.
- Execution of the inference system is handled at the metalevel, by means of a generic search operation that simultaneously takes care of metavariables. Indeed, we have used the same search operation for CCS, for the Hennessy-Milner logic, and for Full LOTOS.
- Using the metalanguage features of Maude, the implementation of the inference system for a logic or language can be extended into an environment for such a logic or language, as we have illustrated in the Full LOTOS case study.

Going back to the CCS case study, we have also seen how the CCS 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 finding all 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 Maude, being so general a framework, can be used for applying higher-order techniques in a first-order framework by means of reflection.

The implementation of the symbolic semantics for Full LOTOS shows how this kind of semantics representations can be extended in two different ways. First, the ACT ONE algebraic specifications are really integrated into the operational semantics, something that really breaks new ground in this approach to executing structural operational semantics. Second, the semantics tool has also been integrated with Full Maude, so that we have provided a Full LOTOS environment with parsing, pretty printing, and execution commands, in which the input/output processing hides from the user the underlying use of Maude.

Other specific-purpose tools, such as the Concurrency Workbench of the New Century (CWB-NC) [20], are more efficient and expressive than our tool. The goal of our prototype is not only to experiment with processes and their capabilities, but also with the semantics itself, represented at a very high mathematical level, by adding or modifying some rules. The design of CWB-NC exploits the language-independence of its analysis routines by localizing language-specific procedures; this enables users to change the system description language by using the Process Algebra Compiler [19], that translates the operational semantics definitions into SML code. We have followed a similar approach, although we have tried to maintain the semantics representation at as high a level as possible, always trying hard not to lose executability. In this way we avoid translating the semantics representations into other languages.

In Section 5.2 we have already cited related work on representations of inference systems in the logical framework Isabelle/HOL. Other logical frameworks and theorem provers have also been used to represent inference systems. The interactive proof development environment Coq [36], based on the Calculus of Constructions extended with inductive types, has been used to represent the π-calculus [35, 33] and the μ-calculus [55] applied to CCS. Coq is used to encode Natural Semantics in [58].

LEGO, an interactive proof development system [39], is used in [62] to formalise a verification system for CCS. It combines theorem proving and model checking by using a theorem prover to reduce or divide the problems to ones which can be model checked.

Typol [23] is a formal language to represent inference rules and operational semantics. Typol programs are compiled into Prolog to create executable type-checkers and interpreters directly from their specifications [22]. Although our implementation is much in the style of logic programming, one advantage over Typol is the possibility of working with user-definable data types and in general algebraic specifications modulo equational axioms. Moreover, in our approach other strategies could be employed besides depth-first search, while keeping the same underlying specification.

XSB [51], a logic programming system that extends Prolog-style SLD resolution with tabled resolution [7], has been used to implement a model checker for value-passing CCS and the modal mu-calculus in [21].

We are currently studying possible extensions of the Interactive Theorem Prover tool (ITP [8, 9]) written in Maude in order to carry (metareasoning) proofs about inductive properties of our operational semantics representations.

Acknowledgments

We are very grateful to José Meseguer, Miguel Palomino, and the anonymous referees for their very helpful comments on earlier versions of this paper which have greatly improved its presentation. We would like to thank David Basin for his very fruitful comments about the Isabelle/HOL foundations, and Carron Shankland for her help and explanations about the LOTOS symbolic semantics.

This paper is a much longer and revised version of [61].

Notes

1. An alternative system supporting rewriting logic computation is ELAN [3] but we do not use it because it does not provide the reflective features that we need in our case studies.
2. The Maude system, including the Full Maude extension, and its documentation are available in Maude's web site at http://maude.cs.uiuc.edu.
3. Due to parsing restrictions, some characters (`[]` `{}` `(),`) have to be preceded by the quote character `‘` when declaring them in Full Maude.
4. The Maude code presented in the rest of this section is declared in a system module `CCS-SEMANTICS` that includes the module `CCS-CONTEXT` above, and which defines the CCS semantics representation (see complete code in [59]).
5. We could easily generalize the restriction operator from $P\backslash l$ to $P\backslash\{l_1, \ldots, l_n\}$, but this would only make the syntax more complex without adding anything new to the semantics representation. The same applies to the relabelling operator.
6. By quoted numbers we mean special quoted identifiers (of sort `Qid`) that have an integer after the quote, like `'3`.
7. In [59] the reader can find these examples as they are introduced in the Maude system; here we use $\bar{t}$ for the metarepresentation of term t. We can also avoid the need to introduce metarepresented terms if we use the parsing and pretty printing capabilities provided by Maude. We have used them for implementing a tool for LOTOS, see Section 6.6.
8. The last rule would be the only one needed if we simplified the rules as described in Section 6.
9. In this definition we also use $\bar{t}$ for the metarepresentation of term t. The reader can find in [59] how these metarepresented terms are written in Maude.
10. Notice that the overbar denotes in this paragraph the complementary label, instead of the metarepresentation.
11. As we have done with the modules and commands in previous sections, since they will be introduced into Full Maude.
12. There is an important separation between the signature used by the users to write their specifications and the abstract syntax we defined in Section 6.2.

References

1. S. Berghofer, "Proofs, programs and executable specifications in higher order logic," PhD thesis, Institut für Informatik, Technische Universität München, 2003.

2. S. Berghofer and T. Nipkow, "Executing higher order logic," in P. Callaghan, Z. Luo, J. McKinna, and R. Pollack (Eds.), *Types for Proofs and Programs: International Workshop, TYPES 2000, Durham, UK*, Selected Papers, Vol. 2277 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002.
3. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. "An overview of ELAN," in Kirchner and Kirchner [38], pp. 329–344.
4. R. Bruni, "Tile logic for synchronized rewriting of concurrent systems," PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999.
5. R. Bruni, J. Meseguer, and U. Montanari, "Internal strategies in a rewriting implementation of tile systems," in Kirchner and Kirchner [38].
6. M. Calder and C. Shankland, "A symbolic semantics and bisimulation for Full LOTOS," in M. Kim, B. Chin, S. Kang, and D. Lee (Eds.), *Proceedings of FORTE 2001, 21st International Conference on Formal Techniques for Networked and Distributed Systems*, Kluwer Academic Publishers, 2001, pp. 184–200.
7. W. Chen and D.S. Warren, "Tabled evaluation with delaying for general logic programs," *Journal of the ACM*, Vol. 43, No. 1, pp. 20–74, 1996.
8. M. Clavel, *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Application*, CSLI Publications, 2000.
9. M. Clavel, "The ITP tool," in A. Nepomuceno, J.F. Quesada, and J. Salguero, (Eds.), *Logic, Language and Information. Proceedings of the First Workshop on Logic and Language*, Kronos, 2001, pp. 55–62.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, "Maude as a metalanguage," in Kirchner and Kirchner [38].
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, *Maude: Specification and Programming in Rewriting Logic*, Computer Science Laboratory, SRI International, 1999. `http://maude.cs.uiuc.edu/maude1/manual.`
12. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, *A Maude Tutorial*, Computer Science Laboratory, SRI International, 2000. `http://maude.cs.uiuc.edu/papers.`
13. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, "Using Maude," in T. Maibaum (Ed.), *Proc. Third Int. Conf. Fundamental Approaches to Software Engineering, FASE 2000, Berlin, Germany, March/April 2000*, Vol. 1783 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 371–374.
14. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, "Maude: Specification and programming in rewriting logic," *Theoretical Computer Science*, Vol.ol. 285, No. 2, pp. 187–243, 2002.
15. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada, "Towards Maude 2.0," in K. Futatsugi (Ed.), *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan,* vol. 36 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2000, pp. 297–318.
16. 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 (Eds.), *FM'99—Formal Methods, Proc. World Congress on Formal Methods in the Development of Computing Systems*, Toulouse, France, September 1999, Vol. II, vol. 1709 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 1684–1703.
17. M. Clavel and J. Meseguer, "Axiomatizing reflective logics and languages," in G. Kiczales (Ed.), *Proceedings of Reflection'96*, 1996, pp. 263–288.
18. M. Clavel and J. Meseguer, "Reflection in conditional rewriting logic," *Theoretical Computer Science*, Vol. 285, No. 2, pp. 245–288, 2002.
19. R. Cleaveland, E. Madelaine, and S.T. Sims, "A front-end generator for verification tools," in Proc. of the Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95), vol. 1019 of *Lecture Notes in Computer Science*, Springer, 1995, pp. 153–173..
20. R. Cleaveland and S.T. Sims, "Generic tools for verifying concurrent systems," *Science of Computer Programming*, Vol. 42, No. 1, pp. 39-47, 2002.
21. B. Cui, Y. Dong, X. Du, K.N. Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, A. Roy-choudhury, S.A. Smolka, and D.S. Warren, "Logic programming and model checking," in C. Palamidessi, H. Glaser, and K. Meinke (Eds.), *Principles of Declarative Programming: 10th International Symposium, PLILP'98. Held*

Jointly with the 6th International Conference, ALP'98, Pisa, Italy, *Proceedings,* vol. 1490 of *Lecture Notes for Computer Science*, Springer, 1998, pp. 1–20.
22. T. Despeyroux, "Executable specification of static semantics," in G. Kahn, D.B. MacQueen, and G.D. Plotkin (Eds.), *Semantics of Data Types,* vol. 173 of *Lecture Notes in Computer Science*, Springer, 1984. pp. 215–233.
23. T. Despeyroux, "TYPOL: A formalism to implement natural semantics," Research Report 94, INRIA, 1988.
24. F. Durán, "A reflective module algebra with applications to the Maude language," PhD thesis, Universidad de Málaga, 1999.
25. H. Eertink, "Executing LOTOS specifications: The SMILE tool," in T. Bolognesi, J. Lagemaat, and C. Vissers (Eds.), *LotoSphere: Software Development with LOTOS*. Kluwer Academic Publishers, 1995.
26. H. Ehrig and B. Mahr, "Fundamentals of algebraic specification 1: Equations and Initial Semantics," EATCS Monographs on Theoretical Computer Science, Springer, 1985.
27. J.C. Fernández, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, "CADP: A protocol validation and verification toolbox," in R. Alur and T. A. Henzinger (Eds.), *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA*, *Proceedings,* vol. 1102 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 437–440.
28. B. Ghribi and L. Logrippo, "A validation environment for LOTOS," in A. Danthine, G. Leduc, and P. Wolper (Eds.), *Protocol Specification, Testing and Verification XIII*, *Proceedings of the IFIP TC6/WG6.1 Thirteenth International Symposium on Protocol Specification, Testing and Verification, Liège, Belgium,* North-Holland, 1993, pp. 93–108.
29. M. Gordon and T. Melham, "Introduction to HOL: A theorem proving environment for higher order logic," Cambridge University Press, 1993.
30. R. Guillemot, M. Haj-Bussein, and L. Logrippo, "Executing large LOTOS specifications," in S. Aggarwal and K. Sabnani (Eds.), *Protocol Specification, Testing, and Verification VIII*, *Proceedings of the IFIP TC6/WG6.1 Eighth International Symposium on Protocol Specification, Testing and Verification*, Atlantic City, USA, North-Holland, 1988, pp. 399–410.
31. M. Hennessy and H. Lin, "Symbolic bisimulations," *Theoretical Computer Science*, Vol. 138, pp. 353–389, 1995.
32. M. Hennessy and R. Milner, "Algebraic laws for nondeterminism and concurrency," *Journal of the ACM*, Vol. 32, No. 1, pp. 137–161, 1985.
33. D. Hirschkoff, "A full formalisation of π-calculus theory in the calculus of constructions," in *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA*, *Proceedings,* vol. 1275 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 153–169.
34. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
35. F. Honsell, M. Miculan, and I. Scagnetto, "π-calculus in (co)inductive-type theory," *Theoretical Computer Science*, Vol. 253, No. 2, pp. 239–285, 2001.
36. G. Huet, G. Kahn, and C. Paulin-Mohring, "The Coq proof assistant: A tutorial: version 7.2," Technical Report 256, INRIA, 2002.
37. ISO/IEC, "LOTOS-A formal description technique based on the temporal ordering of observational behaviour," International Standard 8807, International Organization for standardization—Information Processing Systems—Open Systems Interconnection, Geneva, 1989.
38. C. Kirchner and H. Kirchner (Eds.), *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France*, vol. 15 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 1998.
39. Z. Luo and R. Pollack, "The LEGO proof development system: A user's manual," Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.
40. N. Martí-Oliet and J. Meseguer, "Rewriting logic as a logical and semantic framework," in D. M. Gabbay and F. Guenthner (Eds.), *Handbook of Philosophical Logic, Second Edition*, vol. 9, Kluwer Academic Publishers, 2002, pp. 1–87.
41. N. Martí-Oliet and J. Meseguer, "Rewriting logic: Roadmap and bibliography," *Theoretical Computer Science*, Vol. 285, No. 2, pp. 121–154, 2002.
42. T.F. Melham, "A mechanized theory of the π-calculus in HOL," *Nordic Journal of Computing*, Vol. 1, No. 1, pp. 50–76, 1994.

43. J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, Vol. 96, No. 1, pp. 73–155, 1992.
44. J. Meseguer, "Research directions in rewriting logic," in U. Berger and H. Schwichtenberg (Eds.), *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany*, NATO ASI Series F: Computer and Systems Sciences 165, Springer, 1998, pp. 347–398.
45. J. Meseguer, K. Futatsugi, and T. Winkler, "Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents," in *Proceedings of the 1992 International Symposium on New Models for Software Architecture*, Research Institute of Software Engineering, Tokyo, Japan, 1992, pp. 61–106.
46. R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.
47. M. Nesi, "Mechanising a modal logic for value-passing agents in HOL," in B. Steffen and D. Caucal (Eds.), *Infinity'96, First International Workshop on Verification of Infinite State Systems,* vol. 5 of *Electronic Notes in Theoretical Computer Science. Elsevier*, 1996.
48. M. Nesi, "Formalising a value-passing calculus in HOL," *Formal Aspects of Computing*, Vol. 11, pp. 160–199, 1999.
49. T. Nipkow, L.C. Paulson, and M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic," vol. 2283 of *Lecture Notes in Computer Science*, Springer, 2002.
50. G.D. Plotkin, "A structural approach to operational semantics," Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
51. P. Rao, K. Sagonas, T. Swift, D.S. Warren, and J. Freire, "XSB: A system for efficiently computing well-founded semantics," in J. Dix, U. Furbach, and A. Nerode (Eds.), *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning,* vol. 1265 of *Lecture Notes in Artificial Intelligence*, Springer, 1997, pp. 430–440.
52. C. Röeckl, "A first-order syntax for the π-calculus in Isabelle/HOL using permutations," in S. Ambler, R. Crole, and A. Momigliano (Eds.), *Proc. MERLIN'01*, vol. 58.1 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2001.
53. C. Röeckl, "On the mechanized validation of infinite-state and parameterized reactive, and mobile systems," PhD thesis, Fakultät für Informatik, Technische Universität München, 2001.
54. C. Röeckl, D. Hirschkoff, and S. Berghofer, "Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the π-calculus and mechanizing the theory of contexts," in F. Honsell and M. Miculan (Eds.), *Proc. FOSSACS'01,* vol. 2030 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 364–378.
55. C. Sprenger, "A verified model checker for the modal π-calculus in Coq," in B. Steffen (Ed.), *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal*, *Proceedings,* vol. 1384 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 167–183.
56. 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, 1999.
57. C. Stirling, "Modal and temporal logics for processes," in F. Moller and G. Birtwistle (Eds.), *Logics for Concurrency—Structure versus Automata (8th Banff Higher Order Workshop, August 27–September 3, 1995, Proceedings)*, vol. 1043 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 149–237.
58. D. Terrasse, "Encoding natural semantics in Coq," in V.S. Alagar (Ed.), *Algebraic Methodology and Software Technology, 4th International Conference, AMAST '95, Montreal, Canada*, *Proceedings,* vol. 936 of *Lecture Notes in Computer Science*, Springer, 1995, pp. 230–244.
59. A. Verdejo, "CCS and LOTOS Semantics Implementation in Maude," Web page. `http://www.ucm.es/sip/alberto/ccs-lotos`.
60. A. Verdejo, "LOTOS symbolic semantics in Maude," Technical Report 122-02, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2002. `http://www.ucm.es/sip/alberto`.
61. A. Verdejo and N. Martí-Oliet, "Implementing CCS in Maude," in T. Bolognesi and D. Latella (Eds.), *Formal Methods For Distributed System Development. FORTE/PSTV 2000 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE*

XIII) and Protocol Specification, Testing and Verification (PSTV XX), Pisa, Italy, Kluwer Academic Publishers, 2000, pp. 351–366.

62. S. Yu and Z. Luo, "Implementing a model checker for LEGO," in J. Fitzgerald, C. B. Jones and P. Lucas (Eds.), *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz*, Austria, September 1997), vol. 1313 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 442–458.