Building tools for LOTOS Symbolic Semantics in Maude*

Alberto Verdejo

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

Abstract. We describe a formal tool based on a symbolic semantics for Full LOTOS, where specifications without restrictions in their data types can be executed. The reflective feature 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 operational semantics, to integrate it with ACT ONE specifications, and to build an entire environment with parsing, pretty printing, and input/output processing of LOTOS specifications. Our aim has been to implement a formal tool that can be used by everyone without knowledge of the concrete implementation, but where the semantics representation is at so high level that can be understood and modified by everyone that knows about operational semantics.
Keywords: Full LOTOS, symbolic semantics, rewriting logic, Maude, meta-language.

1 Introduction

The formal description technique LOTOS [15] 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 and CSP, and the mechanism to define and to deal with data types is based on ACT ONE.[1] LOTOS became an international standard (IS-8807) in 1989. Since its standardization, 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.

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 of the tools ignore or restrict the use of data types. Calder and Shankland [3] have defined a *symbolic* semantics for LOTOS which gives meaning to symbolic, or data parameterised processes (see Section 1.1) and avoids infinite branching.

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

[1] 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.

In the last two decades a lot of work has been done regarding LOTOS implementations [13, 12, 10]. In this paper we focus on the use of rewriting logic [19, 20] and particularly the Maude language and system [6] to implement a formal tool based on a symbolic semantics where LOTOS specifications without restrictions in their data types can be executed.

1.1 LOTOS symbolic semantics

The implementation of the LOTOS symbolic semantics given here is entirely based on the work presented in [3]. A symbolic semantics for LOTOS is given by associating a symbolic transition system with each LOTOS behaviour expression P. Following [14], Calder and Shankland define *symbolic transition systems* (STS) as transition systems which separate the data from process behaviour by making the data symbolic. STS are labelled transition systems with variables, both in states and transitions, and conditions which determine the validity of a transition.

Definition 1. *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 $T \xrightarrow{b \quad \alpha} T'$, where α is a simple or structured event and b is a Boolean expression, such that $fv(T') \subseteq fv(T) \cup fv(\alpha)$ and $fv(b) \subseteq fv(T) \cup fv(\alpha)$.*

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

In [3] the intuition and key features of this semantics are presented, together with axioms and inference rules for each LOTOS operator. We will present some of them, together with their representation in Maude, in Section 2.2.

1.2 Rewriting logic and Maude as a metalanguage

Rewriting logic was introduced by Meseguer [19] as a unified model of concurrency in which several well-known models of concurrent systems can be represented in a common framework. Since then much work has been done on the use of rewriting logic as a logical and semantic framework [18], in which many different logics, models of computation, and a wide range of languages, including formal specification languages like LOTOS, can be represented, can be given a precise semantics, and can be executed. Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation [6]. We use in this paper Maude 2.0 [7], a new version with greater generality and expressiveness. In particular, Maude 2.0 allows rewrite conditions which are essential for the LOTOS semantics implementation presented here, as we will see.

Maude should be viewed as a *metalanguage* [5] in which the syntax and semantics of all these models and languages can be formally defined, and in which entire *environments* for such languages can be built (including parsers, execution environments, pretty printing, and input/output). We will see how an environment of this kind has been built for LOTOS in Section 4.

Reflection is the main feature to achieve these powerful metalanguage functionalities. Rewriting logic is reflective [4], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$, and we then have the following equivalence:

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

Intuitively, this means that we can work with theories as *data* at the metalevel, combining and manipulating them, and controlling the rewriting process.

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`, where Maude terms are reified as elements of a data type `Term`, Maude modules are reified as terms in a data type `Module`, the process of reducing a term to normal form is reified by a function `metaReduce`, and the process of applying a rule of a system module to a subject term is reified by a function `metaApply` [6]. These basic operations can be combined to build *strategies* [4] that control the process of rewriting.

If we want to use rewriting logic as a semantic framework and Maude as a metalanguage to *implement* our language, the first thing we have to do is to represent the language $\mathcal{L}$ in question in rewriting logic by a mapping of the form

$$\Phi : \mathcal{L} \longrightarrow RWLogic.$$

In our present case for LOTOS, the map Φ is essentially an identity map, preserving the original structure of the formulas, and mirroring each semantic rule by a corresponding rewrite rule. We will show this representation in Section 2.

Then, we would like to *execute* modules written in that representation. As we will see in Section 2.3 the obtained representation is itself executable, although without values, apart from the predefined Booleans. The LOTOS symbolic semantics is parameterized over the set of *values* and *data expressions*. Thus, if we want to build a usable formal tool, we need more, we need to handle data types, specified using ACT ONE [11].

Instead of defining a data type for representing ACT ONE modules in Maude and operations to represent the reduction process in ACT ONE, we have implemented an automatic translation from ACT ONE modules into functional modules in Maude, and we are then able to use the high-performance Maude reduction engine in a conservative way. We will present in Section 3 this translation, and in Section 3.1 how the modules are extended to be used by the semantics.

In Section 4 we will show how the semantics implementation and the ACT ONE modules translation are integrated to build an entire environment for our formal tool, where LOTOS specifications with complete freedom in their data types and (possibly recursive) process definitions, can be entered and executed by means of a user interface that completely hides the concrete implementation details. We compare in Section 5 our tool with others like the Concurrency Workbench [8] and CADP [16]. Finally, we present some conclusions and future work in Section 6.

2 LOTOS symbolic semantics in Maude

In order to implement the LOTOS symbolic semantics in Maude, we interpret a LOTOS transition $T \xrightarrow{b \quad \alpha} T'$ as a rewriting logic rewrite $T \longrightarrow \{b\}\{\alpha\}T'$. Since the rewriting logic arrow has no labels, we write them as part of the righthand side term.

In this way, the operational semantics rules become conditional rewrite rules where the premises become the rule condition, as shown in Section 2.2, and a transition is possible if and only if the corresponding rewrite can be made in the rewrite theory representing the semantics, as shown in Section 2.2. This method was used in [18] to represent the operational semantics for Milner's CCS, and in [9] to map action semantics into rewriting logic.

By following a different approach, we presented in [24] an implementation of the CCS operational semantics. There, CCS transitions $P \xrightarrow{\alpha} P'$ are represented as terms, and the CCS semantic rules are translated into rewrite rules where the representation of the conclusion is rewritten to the set of representations of the premises. In this way, we start with a transition to be proved valid and work backwards using the rewriting process, maintaining a set of transitions that have to be fulfilled in order to prove the correctness of the initial transition. This transition can be rewritten to the empty set if and only if it is a valid transition in the CCS semantics. This method has been used for representing LOTOS in [22].

The approach followed in this paper has several advantages. The implementation is closer to the mathematical, logical presentation of the semantics. An operational semantics rule establishes that the transition in the conclusion is possible if the transitions in the premises are possible, and that is the interpretation of a conditional rewrite rule with rewrite conditions. The alternative approach needs auxiliary structures like the multisets of statements to be proved valid, which forced us to implement at the metalevel a search strategy that checks if a given multiset can be reduced to the empty set. Now, the necessity of searching appears in the rewrite conditions but the Maude system, which is able to handle these conditions, solves the problem. We will compare these two different approaches again in Section 5.

2.1 LOTOS syntax

There are two different syntax: the concrete syntax used by the specifier (see [23], for full details), and the abstract syntax used by the semantics implementation and introduced in this section. It is defined in the Maude functional module

LOTOS-SYNTAX, which includes DATAEXP-SYNTAX. We use the predefined quoted identifiers to build LOTOS variable, sort, gate, and process identifiers. Booleans are the only predefined data type. LOTOS syntax is extended in a user-definable way when ACT ONE data types specifications are used. Values of these data types will extend the type DataExp below. We will see how it is done in Section 3.

```
fmod DATAEXP-SYNTAX is  protecting QID .
  sort VarId .  op V : Qid -> VarId .
  sort DataExp .
  subsort VarId < DataExp . *** A LOTOS variable is a data expression.
  subsort Bool < DataExp .  *** Booleans are a predefined data type.
endfm

fmod LOTOS-SYNTAX is  protecting DATAEXP-SYNTAX .
  sorts SortId GateId ProcId .
  op S : Qid -> SortId .  op G : Qid -> GateId .  op P : Qid -> ProcId .
  sort BehExp .
  op stop : -> BehExp .
  op exit(_) : ExitParam -> BehExp .
  op _;_ : Action BehExp -> BehExp [prec 35] .
  op _[]_ : BehExp BehExp -> BehExp [prec 40] .
  op _|[_]|_ : BehExp GateIdList BehExp -> BehExp .
  op hide_in_ : GateIdList BehExp -> BehExp [prec 40] .
  [...]
endfm
```

2.2 LOTOS symbolic semantics

First, we define Contexts, which are used to keep the definitions of processes introduced in a LOTOS specification. In order to execute a process instantiation, the process definition has to be looked for in the context. The actual context is built when the LOTOS specification is entered to the tool (we will see how it is done in Section 4.1). In the semantics, a constant context is assumed, and it represents the collection of process definitions. We could say that the semantics is parameterized over this constant, that will be instantiated when a concrete specification is used.

```
fmod CONTEXT is  protecting LOTOS-SYNTAX .
  sort Context .
  op context : -> Context .
  [...]
endfm
```

Now, we can implement the LOTOS symbolic semantics. First we show some operations used in the semantics definition that present problems when implementing them, and how we have solved these problems in Maude.

In the semantics, a set **new-var** of fresh variable names is assumed. As said in [3], strictly speaking, any reference to this set requires a context, i.e.

the variable names occurring so far. Instead of messing up the implementation with this other context, we have preferred to use a predefined Maude utility imported from module `ORACLE`, where a constant `NEWQ` is defined. Each time `NEWQ` is rewritten, it is rewritten to a different quoted identifier. With the following definition, we have the set of fresh variable names.

```
op new-var : -> VarId .
eq new-var = V(NEWQ) .
```

A (data) substitution is written as $[z/x]$ where z is substituted for x. It seems to be easy to implement equationally, and we present below some equations showing how the substitution operation distributes over the syntax of behaviour expressions. However, if we want to allow user-definable data expressions by means of an ACT ONE specification, we cannot completely define this operation now, because we do not know at this point the syntax of data expressions. We will describe in Section 3.1 how the module containing the new syntax is automatically extended to define this operation on new data expressions.

```
op _[_/_] : BehExp DataExp VarId -> BehExp .
op _[_/_] : DataExp DataExp VarId -> DataExp .
eq stop [E' / E] = stop .
eq g ! E1 ; P [E' / E] = g ! (E1[E' / E]) ; (P[E' / E]) .
eq P1 [] P2 [E' / E] = (P1[E' / E]) [] (P2[E' / E]) .
[...]
```

Also a function *vars* is used to obtain the variables occurring in a behaviour expression. And we have the same problem, that is, we cannot define it completely at this level since data expressions syntax is user-definable. We will see in Section 3.1 how it is defined automatically for new data expressions.

```
op vars : BehExp -> VarSet .
op vars : DataExp -> VarSet .
eq vars(stop) = mt .
eq vars(g ? x : S ; P) = x U vars(P) .
eq vars(P1 [] P2) = vars(P1) U vars(P2) .
eq vars(x) = x .
[...]
```

Now we can define the symbolic transitions. As said above, a transition $T \xrightarrow{b \quad \alpha} T'$, where T and T' are behaviour expressions, b is a transition condition, and α is an event, will be represented as a rewrite $T \longrightarrow \{b\}\{\alpha\}T'$, where the righthand side term is of sort `TCondEventBehExp`.

```
sort TCondEventBehExp .
op {_}{_}_ : TransCond Event BehExp -> TCondEventBehExp .
```

We present some LOTOS symbolic semantics rules and their representation as rewrite rules in Table 1 (the complete set of rules can be found in [23]). We also show the inference rules to ease the comparison between the mathematical and

Maude representations. The inference rules are not exactly the ones presented in [3] because we have generalized them to allow multiple event offers at an action.

The rules for the prefix operator show how axioms are represented as rewrite rules without conditions. The choice range rule shows how nondeterministic elections can be made by using rewrite rules. The hide rules show how side conditions in the inference rules are added as conditions in the rewrite rules. Finally, the general parallelism rule shows how external definitions can be used, as the one defining the substitution (not shown in figure).

After having implemented all the semantics rules for behaviour expressions, we have the following conservativity result: Given a LOTOS behaviour expression P, there are a transition condition b, an event a, and a behaviour expression P' such that

$$P \xrightarrow{b \quad a} P'$$

if and only if `P` can be rewritten into `{b}{a}P'` using the presented rules.

In [3], it is also defined the concept of a *term*, which consists of an STS, T, paired with a substitution, σ, and written as T_σ. Transitions between terms are also defined. We have implemented these transitions as we have done for behaviour expressions (see [23]).

2.3 Execution example

By using the `search` Maude command, that looks for all the rewrites of a given term that match a given pattern, we can find all the possible transitions of a behaviour expression.

```
Maude> search
  G('g) ; G('h) ; stop
|[ G('g) ]|
  (G('a) ; stop [] G('g) ; stop) => X:TCondEventBehExp .
Solution 1 (state 1)
X:TCondEventBehExp --> {true}{G('a)}G('g) ; G('h) ; stop |[G('g)]| stop
Solution 2 (state 2)
X:TCondEventBehExp --> {true}{G('g)}G('h) ; stop |[G('g)]| stop
No more solutions.

Maude> search G('h) ; stop |[G('g)]| stop => X:TCondEventBehExp .
Solution 1 (state 1)
X:TCondEventBehExp --> {true}{G('h)}stop |[G('g)]| stop
No more solutions.
```

But we cannot use data expressions, apart from the predefined Booleans, because we have not introduced any ACT ONE specification, and we have to write identifiers using the abstract syntax. But these specifications are part of a Full LOTOS specification, and therefore, user-definable. We will see in the following sections how we give semantics to ACT ONE specifications and how they can be integrated with the previous LOTOS semantics implementation.

prefix axioms

$$a; P \xrightarrow{\mathrm{tt} \quad a} P$$

$$g\ d_1 \ldots d_n; P \xrightarrow{\mathrm{tt} \quad g E'_1 \ldots E'_n} P$$

$$g\ d_1 \ldots d_n[SP]; P \xrightarrow{SP \quad g E'_1 \ldots E'_n} P$$

where $E'_i = \begin{cases} E_i \text{ if } d_i &= !E_i \\ x_i \text{ if } d_i &= ?x_i : S_i \end{cases}$

```
rl A ; P => {true}{A}P .
rl g O ; P => {true}{g eOff(O)}P .
rl g O [SP] ; P => {SP}{g eOff(O)}P .

op eOff : Offer -> EOffer .
eq eOff(! E) = E .
eq eOff(? x : S) = x .
eq eOff(O O') = eOff(O) eOff(O') .
```

choice range rules

$$\frac{P[g_i/g] \xrightarrow{b \quad \alpha} P'}{\textbf{choice } g \textbf{ in } [g_1, \ldots, g_n] \, [\,] \, P \xrightarrow{b \quad \alpha} P'}$$

for each $g_i \in \{g_1, \ldots, g_n\}$

```
crl choice g in [GIL][] P => {b}{a}P'
 if select(GIL) => gi  /\  P[gi / g] => {b}{a}P' .
sort GateId? . subsort GateId < GateId? .
op select : GateIdList -> GateId? .
rl select(g) => g .
rl select(g, GIL) => g .
rl select(g, GIL) => select(GIL) .
```

hide rules

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\textbf{hide } g_1, \ldots, g_n \textbf{ in } P \xrightarrow{b \quad \mathbf{i}} \textbf{hide } g_1, \ldots, g_n \textbf{ in } P'} \quad \textbf{name}(\alpha) \in \{g_1, \ldots, g_n\}$$

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\textbf{hide } g_1, \ldots, g_n \textbf{ in } P \xrightarrow{b \quad \alpha} \textbf{hide } g_1, \ldots, g_n \textbf{ in } P'} \quad \textbf{name}(\alpha) \notin \{g_1, \ldots, g_n\}$$

```
crl hide GIL in P => {b}{i}hide GIL in P'
 if P => {b}{a}P'  /\  (name(a) in GIL) .
crl hide GIL in P => {b}{a}hide GIL in P'
 if P => {b}{a}P'  /\  not(name(a) in GIL) .
```

general parallelism rules (not synchronising)

$$\frac{P_1 \xrightarrow{b \quad \alpha} P'_1}{P_1 |[g_1, \ldots, g_n]| P_2 \xrightarrow{b\sigma \quad \alpha\sigma} P'_1\sigma |[g_1, \ldots, g_n]| P_2} \quad \textbf{name}(\alpha) \notin \{g_1, \ldots, g_n, \delta\}$$

where $\alpha = gE_1 \ldots E_n$, $\sigma = \sigma_1 \ldots \sigma_n$, $dom(\sigma_i)$ are disjoint, and
$\sigma_i = \begin{cases} [z_i/x_i] \text{ if } E_i = x_i,\ x_i \in vars(P_2) \text{ and } z_i \in \textbf{new-var}. \\ [\,] \qquad \text{otherwise} \end{cases}$

```
crl P1 |[GIL]| P2 => {b sp(a,vars(P2))}{a sp(a,vars(P2))}
                     ((P' sp(a,vars(P2))) |[GIL]| P2)
 if P1 => {b}{a}P' /\ not(name(a) in (GIL, delta)) .
```

Table 1. Some semantics rules, and their implementation in Maude.

3 ACT ONE modules translation

We want to be able to introduce into our tool ACT ONE specifications, which will be then translated internally to Maude functional modules.

Thus, we have to define ACT ONE syntax first. In Maude, the *syntax definition* for a 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 [5]. This is specially important when $\mathcal{L}$ has user-definable syntax, as it is our case with ACT ONE. The grammar of ACT ONE can be found in [23].

After having defined the module with ACT ONE syntax, we can use function `metaParse` from `META-LEVEL`, which receives as arguments the representation of a module M and the representation of a list of tokens and it returns the metarepresentation of the parsed term (a parse tree that may have bubbles) of that list of tokens for the signature of M.

The next step consists in defining an operation `translate` that receives the parsed term and returns a functional module with the same semantics as the introduced ACT ONE specification.

$$\texttt{QidList} \xrightarrow{\texttt{metaParse}} \texttt{Grammar}_{\text{ACT ONE}} \xrightarrow{\texttt{translate}} \texttt{FModule}$$

Notice that we start with a `QidList` (a list of quoted identifiers). It is obtained from the user input (see Section 4.2).

With our translation we achieve the following result: Given an ACT ONE specification SP, and terms t and t' in SP, we have

$$SP \models t \equiv t' \iff M \models t_M \equiv t'_M$$

where $M = \texttt{translate}(\texttt{metaParse}(\texttt{ACTONE-GRAMMAR},SP))$, and t_M and t'_M are the representations of t and t' in M.

We do not show here the implementation of the translation (it can be found in [23]), but we show an example in Figure 1. Notice that 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 type 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 data expressions of this type). This is done in this way because we want to integrate ACT ONE modules with LOTOS specifications, but the translation is useful by itself, since it provides us with a tool in Maude where ACT ONE specifications can be entered and executed.

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

```
fmod Naturals is
  including DATAEXP-SYNTAX .
  sorts Nat .
  subsort VarId < Nat .
  subsort Nat < DataExp .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  eq 0 + x:Nat = x:Nat .
  eq s(x:Nat) + y:Nat =
     s(x:Nat + y:Nat) .
endfm
```

Fig. 1. The ACT ONE specification on the left is translated into the functional module on the right.

3.1 Module extensions

In Section 2.2 we saw that the operation that performs the syntactic substitution and the operation that extracts the variables occurring in a behaviour expression were not completely defined. The reason why we cannot define them completely when defining the semantics is the same in both cases: the presence of data expressions with user-definable syntax.

Now that we know the ACT ONE specification and we have translated it to a functional module, we can define these operations on data expressions using the new syntax. Due to the metaprogramming features of Maude, we can do it automatically. We have defined operations that take a module M and return the same module M but where equations defining the substitution and extraction of variables over expressions built using the signature in M have been added.

For example, if the operation `addOperVars` is applied to the module `Naturals` above, it adds the following equations:[2]

```
eq vars(0) = mt .
eq vars(s(v1:Nat)) = vars(v1:Nat) .
eq vars(v1:Nat + v2:Nat) = vars(v1:Nat) U vars(v2:Nat) .
```

4 Building the user interface for LOTOS

We want to implement a formal tool where complete LOTOS specifications (with an ACT ONE data types specification, a main behaviour expression, and process definitions) are entered and executed. In order to *execute* or *simulate* the specification, we want to be able to traverse the symbolic transition system generated for the main behaviour expression by using the symbolic semantics instantiated

[2] In Maude 2.0 a variable is an identifier composed of a name, followed by a colon, followed by a sort name. In this way, variables do not have to be declared in variable declarations, although they are still allowed for convenience.

with the data types given in ACT ONE and the given process definitions. We present here only the main ideas used in our implementation; full details can be found in [23].

The following module defines the commands of our tool.

```
fmod LOTOS-TOOL-SIGN is  protecting LOTOS-SIGN .
  sort LotosCommand .
  op show process . : -> LotosCommand .
  op show transitions . : -> LotosCommand .
  op show transitions of_. : BehExp -> LotosCommand .
  op cont_. : MachineInt -> LotosCommand .
  op cont . :  -> LotosCommand .
  op show state . : -> 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. Command `cont` is a shorthand for `cont 1`. The sixth command is used to show the current *state* of execution, that is, the current condition, trace and possible next transitions.

4.1 Tool state handling

In our tool, the persistent state of the system is given by a single object which maintains the tool state. This object has the following attributes:

- `semantics`, to keep the actual module where behaviour expressions can be executed, that is, the module `LOTOS-SEMANTICS` in Section 2.2 extended with the syntax and semantics for new data expressions;
- `lotosProcess`, to keep the behaviour expression that labels the node in the symbolic transition system that has been reached during the execution;
- `transitions`, to keep the set of possible transitions from `lotosProcess`;
- `trace`, to keep the sequence of events performed in the path from the root of the STS to the current node;
- `condition`, to keep the conjuntion of transition conditions in that path; and
- `input` and `output`, to handle the communication with the user.

We declare the following class by using the notation for classes in object-oriented modules [6]:

```
class ToolState | semantics : Module, lotosProcess : Term,
                  transitions : TermSeq, trace : Term, condition : Term,
                  input : QidList, output : QidList .
```

Then we describe by means of rewrite rules the behaviour of the tool when a LOTOS specification or the different commands are entered into the system. For example, there is a rule which processes a LOTOS specification entered to the system. We allow LOTOS specifications with four arguments: the name of the specification, an ACT ONE specification defining the data types to be used, the main behaviour expression, and a list of process definitions (either the ACT ONE specification or the list of processes can be empty). No local declarations are allowed. When a specification is entered, the `semantics` attribute is set to a new module built as follows: first, the ACT ONE part of the specification is translated to a functional module; then, equations defining the extraction of variables and substitution are added (as explained in Section 3.1); the resulting module is joined with the metarepresentation of module `LOTOS-SEMANTICS`; and, finally, an equation defining the constant `context` (Section 2.2) with the definitions of processes given in the specification is added. The `lotosProcess` attribute is also updated to the behaviour expression in the introduced specification (after having converted it to a term of sort `BehExp`), and the rest of attributes are initialized.

Tool commands are handled by rules as well. For example, there is a rule that handles the `show transitions` command. It modifies the `transitions` attribute by using an operation which receives a module with the semantics implementation (extended with the syntax and semantics of data expressions) and a term t representing a behaviour expression, and returns the sequence of terms representing the possible transitions of t. It uses the operation `metaSearch` that represents at the metalevel the `search` command used in Section 2.3.

4.2 The LOTOS tool environment

Input/output of specifications and of commands is accomplished by the predefined module `LOOP-MODE` [6], 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). We have complete flexibility for defining this state. In our tool we use an object of the `ToolState` class. When something is written in the Maude prompt enclosed in parentheses it is placed in the first slot of the loop object, as a list of quoted identifiers. Then it is parsed by using the adequate grammar, and the parsed term is put in the `input` attribute of the tool state object. Finally, the rules describing the tool state handling process it. 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.

4.3 Execution example

This is an example of an interaction with the LOTOS tool. Although we use here a very simple example, we have used the tool to execute larger examples [23], including the Alternating Bit Protocol and the Sliding Window Protocol (with more than 550 lines of code) [21]. Our tool has proved itselt quite practical, giving the answer to the entered commands in few milliseconds.

```
Maude> (specification SPEC
type Naturals is
  [as shown above]
endtype
behaviour
 h ! 0 ; stop [] (  g ! (s(0)) ; stop
                  |[ g ]|
                    g ? x : Nat ; h ! (x + s(0)) ; stop )
endspec)

Maude> (show transitions .)
Trace : nil
Condition : true
TRANSITIONS :
1.  {true}{h 0}stop
2.  {x = s(0)}{g s(0)}stop |[g]| h ! s(s(0)); stop

Maude> (cont 2 .)
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 .
```

5 Comparison with other tools

As we said in the introduction of Section 2, we have implemented a similar tool by using a different approach, where transitions are represented as terms [22]. The main differences (besides the semantics representation, which is quite different) are found in the things that are done in the object level (level of the semantics representation) and the metalevel (by using reflection). In [22], a search operation defined at the metalevel is used to check if a transition is possible. It traverses a tree with all the possible rewrites of a term, moving continuously between the object level and the metalevel. In the implementation described in this paper, the search occurs completely at the object level, which makes it quite faster (and simpler). The fact of moving continuosly between the two levels allows us in [22] to define more things at the metalevel, like the substitution operation and extraction of variables, defined used the syntax of `Terms` at the metalevel. Here we follow a different approach as explained in Section 3.1, which we think is much more elegant and more general.

The Concurrency Workbench of the New Century (CWB-NC) [8] is an automatic verification tool where systems in several specification languages can be executed and analyzed. Regarding LOTOS, CWB-NC accepts Basic LOTOS, because it does not support value-passing process algebras. The design of the system exploits the language-independence of its analysis routines by localizing language-specific procedures, which enables users to change the system description language by using the Process Algebra Compiler, 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 level as possible, although being executable. We have also implemented the semantics of the Hennessy-Milner modal logic for CCS and the subset of FULL [2] corresponding to this logic for LOTOS. Both implementations follow the same idea, using an operation to calculate the one-step successors of a process which in its turn uses the operational semantics definitions. Thus, the implementation of the analyzing algorithm, that is, the representation in Maude of the modal logic semantics, is the same in both cases, obtaining similar achievements than in the CWB-NC on keeping separated the language-specific features from the general ones.

The Caesar/Aldebaran Development Package (CADP) [16] is a toolbox for protocol engineering, with several functionalities, from interactive simulation (as we do in our tool) to formal verification. In order to support different specification languages, CADP uses low-level intermediate representations, which forces the implementer of a new semantics to write compilers that generate these representations. CADP has already been used to implement FULL [1], although with the severe restrictions to finite types and to the standard semantics of LOTOS instead of the symbolic one.

6 Conclusions and future work

We have presented a new example of how rewriting logic and Maude can be used as a semantic framework and metalanguage, where entire environments and tools for the execution of formal specification languages can be built. In this process, reflection plays a decisive role.

We have implemented the LOTOS symbolic semantics in Maude by representing transitions as rewrites and by representing the semantics rules as conditional rewrite rules where the transitions in the premises become the conditions. This approach is different from the one used in [22, 24], and presents several advantages as explained in Sections 2 and 5.

In addition, we have implemented a translation from ACT ONE specifications to Maude functional modules, which allows as to execute these specifications in Maude by using its high-performance reduction engine. These functional modules are integrated with the semantics, obtaining an implementation of the LOTOS symbolic semantics with user-definable data types.

Finally, we have implemented in Maude a user interface for our tool that allows the user not to use Maude directly, but instead a tool built on Maude whose input is a LOTOS specification and where the specification can be executed by means of commands that traverse the corresponding labelled transtition system.

Based on the symbolic semantics used in this paper, a symbolic bisimulation [3] and a modal logic FULL [2] have been defined. We plan to extend our tool so that we can check if two processes are bisimilar, or if a process satisfies a given modal logic formula. We have already implemented a subset of FULL without data values, and we have integrated it with our tool. The part of the logic with data values deserves more study, and we think that some kind of theorem proving will be needed. Rewriting logic and Maude have been proved highly valuable also for these subjects [4].

Acknowledgements

I would like to thank Carron Shankland for her helpful answers about LOTOS symbolic semantics. I am also very grateful to Narciso Martí-Oliet for his useful comments and suggestions on earlier versions of this paper.

References

1. J. Bryans and C. Shankland. Implementing a modal logic over data and processes using XTL. In Kim et al. [17], pages 201–218.
2. M. Calder, S. Maharaj, and C. Shankland. An adequate logic for Full LOTOS. In J. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 384–395. Springer-Verlag, 2001.
3. M. Calder and C. Shankland. A symbolic semantics and bisimulation for Full LOTOS. In Kim et al. [17], pages 184–200.
4. M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Publications, 2000.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude as a metalanguage. In C. Kirchner and H. Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. http://www.elsevier.nl/locate/entcs/volume15.html.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic.* Computer Science Laboratory, SRI International, Jan. 1999. http://maude.csl.sri.com/manual.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 297–318. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume36.html.
8. R. Cleaveland and S. T. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, Jan. 2002.
9. C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In T. Rus, editor, *AMAST: 8th International Conference on Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, pages 407–421. Springer-Verlag, 2000.

10. H. Eertink. Executing LOTOS specifications: the SMILE tool. In T. Bolognesi, J. Lagemaat, and C. Vissers, editors, *LotoSphere: Software Development with LOTOS*. Kluwer Academic Publishers, 1995.
11. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.
12. B. Ghribi and L. Logrippo. A validation environment for LOTOS. In A. Danthine and G. Leduc, editors, *Protocol Specification, Testing, and Verification XIII*, pages 93–108. Norht-Holland, 1993.
13. R. Guillemot, M. Haj-Hussein, and L. Logrippo. Executing large LOTOS specifications. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 399–410. North-Hollland, 1988.
14. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.
15. 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, Sept. 1989.
16. J. -C. Fernandez, 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, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer-Verlag, 1996.
17. M. Kim, B. Chin, S. Kang, and D. Lee, editors. *Proceedings of FORTE 2001, 21st International Conference on Formal Techniques for Networked and Distributed Systems*. Kluwer Academic Publishers, 2001.
18. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, Aug. 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic, Second Edition, Volume 9*. Kluwer Academic Publishers, 2002. http://maude.csl.sri.com/papers.
19. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
20. J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, NATO ASI Series F: Computer and Systems Sciences 165, pages 347–398. Springer-Verlag, 1998.
21. K. Turner. *Using Formal Description Techniques – An Introduction to Estelle, LOTOS and SDL*. John Wiley and Sons Ltd., 1992.
22. A. Verdejo. LOTOS symbolic semantics in Maude. Technical Report 122-02, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Jan. 2002.
23. A. Verdejo. A tool for Full LOTOS in Maude. Technical Report 123-02, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Apr. 2002. http://dalila.sip.ucm.es/~alberto.
24. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In T. Bolognesi and D. Latella, editors, *Formal Methods For Distributed System Development. FORTE/PSTV 2000*, pages 351–366. Kluwer Academic Publishers, 2000.