A tool for Full LOTOS in Maude[*]

Alberto Verdejo

Technical Report 123-02
Dpto. Sistemas Informáticos y Programación
Universidad Complutense de Madrid. Spain

20th April 2002

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

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.

Contents

1 Introduction

The formal description technique LOTOS [16] 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 [22] and CSP [15], and the mechanism to define and to deal with data types is based on ACT ONE [13].[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, 4] have defined a *symbolic* semantics for LOTOS which gives meaning to symbolic, or data parameterised processes (see Section 1.1) and avoids infinite branching.

In this paper we use rewriting logic [20, 21] and Maude [7] to implement a formal tool based on this 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, 4]. 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 *(Symbolic Transition Systems)*
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 $fv(T') \subseteq fv(T) \cup fv(\alpha)$, $fv(b) \subseteq fv(T) \cup fv(\alpha)$ and $\#(fv(\alpha) - fv(T)) \leq 1$.*

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

In [4] the intuition and key features of this semantics are presented, together with axioms and inference rules for each LOTOS operator. We will see 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 [20] 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 [19], in

[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.

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 [7]. We use in this paper Maude 2.0 [8], a new version with greater generality and expressiveness. In particular, Maude 2.0 allows rewrite conditions which are essential for the shown LOTOS semantics implementation.

Maude should be viewed as a *metalanguage* [6] 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 5.

Reflection is the main feature to achieve these powerful metalanguage functionalities. Rewriting logic is reflective [5], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$, and we then have the following equivalence:

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

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`, where Maude terms are reified as elements of a data type `Term`, 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` [7]. These basic operations can be combined to build *strategies* [9] 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

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

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 Sections 2.1 and 2.2.

Then, we would like to *execute* 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 [13].

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. We will present in Section 4 this translation, and in Section 4.1 how the modules are extended to be used by the semantics.

In Section 5 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 6 our tool with others like the Concurrency Workbench [10] and CADP [17]. Finally, we present some conclusions and future work in Section 7.

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, Theorem 1. This method was used in [19] to represent the operational semantics for Milner's CCS, and in [11] to map action semantics into rewriting logic.

By following a different approach, we presented in [25] 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 [24].

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.

2.1 LOTOS syntax

There are two different syntax: the concrete syntax used by the specifier (see Section 4), 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 4.

```
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.
  subsort VarId < Bool .

  sort DataExpList .
  subsort DataExp < DataExpList .
  op nilDEL : -> DataExpList .
  op _,_ : DataExpList DataExpList -> DataExpList [assoc prec 35] .
endfm

fmod LOTOS-SYNTAX is
  protecting DATAEXP-SYNTAX .

  *** identifiers contructors
  sorts SortId GateId ProcId .

  op S : Qid -> SortId .
  op G : Qid -> GateId .
  op P : Qid -> ProcId .
```

The basic processes in LOTOS are inaction (deadlock), and successful termination possibly with data values. The exit paramenter can be a value or a variable over a set of values. For simplicity, as it is done in [4], we will assume that only one parameter can occur at an `exit` behaviour expression.

```
  sort BehaviourExp .

  op stop : -> BehaviourExp [format (b! o)] .
  op exit : -> BehaviourExp .
  op exit(_) : ExitParam -> BehaviourExp .

  sort ExitParam .
  subsort DataExp < ExitParam .

  op any_ : SortId -> ExitParam .
```

Actions occur at gates, and may or may not have data offers associated with them. We have remove the restriction used in [4] that only one event offer can occur at an action, allowing multiple offers. Actions may also be subject to *selection predicates*, which are Boolean conditions that may restrict the communicated data values. The special action `i` is the (unobservable) internal event.

```
  sorts SimpleAction StrucAction Action Offer SelecPred IdDecl .

  subsort GateId < SimpleAction .
  subsorts SimpleAction StrucAction < Action .

  op i : -> SimpleAction .
  op __ : GateId Offer -> StrucAction [prec 30 gather(e e)] .

  op !_ : DataExp -> Offer [prec 25] .
  op ?_ : IdDecl -> Offer [prec 25] .
  op _:_ : VarId SortId -> IdDecl [prec 20] .

  op __ : Offer Offer -> Offer [assoc prec 27] .

  op _[_] : SimpleAction SelecPred -> Action [prec 30] .
  op _[_] : StrucAction SelecPred -> Action [prec 30] .

  subsort Bool < SelecPred .
```

Actions and processes are combined using the following operators:

```
  *** action prefixing
  op _;_ : Action BehaviourExp -> BehaviourExp [frozen prec 35] .

  *** choice
  op _[]_ : BehaviourExp BehaviourExp -> BehaviourExp [assoc frozen prec 40] .
  op choice_[]_ : IdDecl BehaviourExp -> BehaviourExp [frozen prec 40] .
  op choice_in[_][]_ : GateId GateIdList BehaviourExp -> BehaviourExp [frozen prec 40] .

  *** parallelism
  sort GateIdList .
  subsort GateId < GateIdList .
  op nilGIL : -> GateIdList .
  op ALL : -> GateIdList .
  op _,_ : GateIdList GateIdList -> GateIdList [assoc prec 35] .

  eq nilGIL, GIL:GateIdList =  GIL:GateIdList .
  eq GIL:GateIdList, nilGIL =  GIL:GateIdList .
  eq g:GateId, ALL = ALL .
  eq ALL, g:GateId = ALL .

  sort ParOp .
  op |[_]| : GateIdList -> ParOp .
  op || : -> ParOp .
  op ||| : -> ParOp .
  op ___ : BehaviourExp ParOp BehaviourExp -> BehaviourExp [frozen prec 40] .
  op par_in[_]__ : GateId GateIdList ParOp BehaviourExp ->
               BehaviourExp [frozen prec 40] .

  *** enable
  op _>>_ : BehaviourExp BehaviourExp -> BehaviourExp [frozen prec 40] .
  op _>> accept_in_ : BehaviourExp IdDecl BehaviourExp -> BehaviourExp [frozen prec 40] .

  *** disable
  op _[>_ : BehaviourExp BehaviourExp -> BehaviourExp [frozen prec 40] .

  *** guard
  op [_]->_ : SelecPred BehaviourExp -> BehaviourExp [frozen prec 40] .

  *** hide
  op hide_in_ : GateIdList BehaviourExp -> BehaviourExp [frozen prec 40] .

  *** variable declaration
  op let_=_in_ : VarId DataExp BehaviourExp -> BehaviourExp [prec 40] .
```

In the abstract syntax a process instantiation always has a gate identifier list and a data expression list, either of which may be empty.

```
  *** process instantiation
  op _[_](_) : ProcId GateIdList DataExpList -> BehaviourExp [frozen prec 30] .

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 5.4). 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 .

  sorts ProcDef Context Context? .
  subsorts ProcDef < Context < Context? .

  op process : ProcId GateIdList DataExpList BehaviourExp -> ProcDef .

  op nil : -> Context .
  op _&_ : Context? Context? -> Context? [assoc comm id: nil prec 42] .

  op _definedIn_ : ProcId Context -> Bool .
  op def : Context ProcId -> Context? .
  op not-defined : -> Context? .

  op context : -> Context .

  vars X X' : ProcId .
  var P : BehaviourExp .
  var GIL : GateIdList .
  var DEL : DataExpList .
  var C : Context .
```

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 `Context?` are well-formed contexts (of sort `Context`).

```
  cmb process(X,GIL,DEL,P) & C : Context if not(X definedIn C) .

  eq X definedIn nil = false .
  eq X definedIn (process(X',GIL,DEL,P) & C) = (X == X') or (X definedIn C) .
```

The evaluation of `def(C,X)` returns the process definition associated to process identifier `X` if it exists; otherwise, it returns the wrong context `not-defined`.

```
  eq def(nil, X) = not-defined .
  eq def(process(X,GIL,DEL,P) & C, X) = process(X,GIL,DEL,P) .
  ceq def(process(X',GIL,DEL,P) & C, X) = def(C, X) if X =/= X' .
endfm
```

Now, we can implement the LOTOS symbolic semantics. First, we define the elements of a symbolic transition, that is, events and transition conditions.

```
mod LOTOS-SEMANTICS is
  pr CONTEXT .
  pr ORACLE .

  sorts SimpleEv StructEv EOffer Event TransCond .
  subsort SimpleAction < SimpleEv .
  subsorts SimpleEv StructEv < Event .
  subsort DataExp < EOffer .

  op delta : -> GateId .
  op __ : GateId EOffer -> StructEv [prec 30 gather(e e)] .

  op __ : EOffer EOffer -> EOffer [assoc prec 27] .

  *** Trace of events (used for the tool output)

  sort Trace .
  subsort Event < Trace .
  op nil : -> Trace .
  op __ : Trace Trace -> Trace [assoc id: nil prec 35 gather(e E)] .

  *** transition conditions

  subsort Bool < TransCond .
  op _=_ : DataExp DataExp -> TransCond [prec 25] .
  op _/\_ : TransCond TransCond -> TransCond [assoc comm] .

  eq true /\ b = b .
```

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

```
  sort TCondEventBehExp .
  subsort BehaviourExp < TCondEventBehExp .
  op {_}{_}_ : TransCond Event TCondEventBehExp -> TCondEventBehExp [frozen] .
```

In this semantic representation, the rewrite rules have the property of being sort-increasing, i.e., in a rewrite $t \longrightarrow t'$, the least sort of t' is bigger than the least sort of t. Thus, one rule cannot be applied unless the resulting term is well formed. For example, although `A ; P` $\longrightarrow$ `{true}{A}P` is a correct transition, we cannot derive `(A ; P) || Q` $\longrightarrow$ `({true}{A}P) || Q` because the right-hand side term is not well formed.

But the sort-increasing mechanism is not enought if we have rewrite conditions. If we have the rewrite condition `P => {b}{a}Q`, then `P` is tried to be rewritten in any possible way, and the result is matched against the pattern `{b}{a}Q`. If in a concrete application `P` is `(A ; P') || Q'`, it is also rewritten to `({true}{A}P') || Q'` although then the result is rejected. The problem appears when we have recursive processes, because the search that tries to satisfy the rewrite condition can become infinite. For example, if `P'` above is recursive, `P' = A ; P'`, then `P` is rewritten to `({true}{A}P') || Q'`, `({true}{A}{true}{A}P') || Q'`, etc., although all the results are rejected because they are not well formed. Our solution to this problem has been to declare the syntax operators and the operator `{_}{_}_` as `frozen`, which prevents an operators arguments from being rewritten by rules. Note that using this attribute effectively changes the denotational as well as the operational semantics of the frozen operator by disallowing the congruence proof rule.

```
  vars B B' : Bool .
```

```
var A : SimpleAction .
vars E E' E1 E2 : DataExp .
vars g g' g'' gi : GateId .
vars O O' : Offer .
vars EO EO1 EO2 : EOffer .
var SP : SelecPred .
vars x y z : VarId .
var S : SortId .
var p : ProcId .
vars GIL GIL' : GateIdList .
var OP : ParOp .
vars DEL DEL' : DataExpList .
vars b b' b1 b2 : TransCond .
vars a a' a1 a2 : Event .
vars P P' P1 P2 P1' P2' : BehaviourExp .
var N : MachineInt .
```

Before defining the semantic rules, we define several functions used by the semantics. In the semantics, a set **new-var** of fresh variable names is assumed. As said in [4], 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.

```
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 the 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 4.1 how the module containing the new syntax is automatically extended to define this operation on new data expressions.

```
sort Substitution .
op `[`] : -> Substitution .
op `[_/_`] : DataExp DataExp -> Substitution .
op `[_/_`] : GateId GateId -> Substitution .

op __ : Substitution Substitution -> Substitution .

var SU : Substitution .

eq B ([ E' / E ] SU) = (B [ E' / E ]) SU .
eq E ([ E' / E ] SU) = (E [ E' / E ]) SU .
eq DEL ([ E' / E ] SU) = (DEL [ E' / E ]) SU .
eq g ([ E' / E ] SU) = (g [ E' / E ]) SU .
eq O ([ E' / E ] SU) = (O [ E' / E ]) SU .
eq GIL ([ E' / E ] SU) = (GIL [ E' / E ]) SU .
eq P ([ E' / E ] SU) = (P [ E' / E ]) SU .
eq b ([ E' / E ] SU) = (b [ E' / E ]) SU .
eq a ([ E' / E ] SU) = (a [ E' / E ]) SU .
eq EO ([ E' / E ] SU) = (EO [ E' / E ]) SU .
```

```
op __ : Bool Substitution -> Bool .
op __ : DataExp Substitution -> DataExp .
op __ : DataExpList Substitution -> DataExpList .
op __ : GateId Substitution -> GateId .
op __ : Offer Substitution -> Offer .
op __ : GateIdList Substitution -> GateIdList .
op __ : BehaviourExp Substitution -> BehaviourExp .
op __ : TransCond Substitution -> TransCond .
op __ : Event Substitution -> Event .
op __ : EOffer Substitution -> EOffer .

eq P [] = P .
eq b [] = b .
eq a [] = a .
eq E [] = E .

eq x [ E' / E ] = if (x == E) then E' else x fi .
eq true [ E' / E ] = true .
eq false [ E' / E ] = false .
eq not(B) [ E' / E ] = not(B[ E' / E ]) .
eq (B and B') [ E' / E ] = (B [ E' / E ]) and (B' [ E' / E ]) .
eq (B or B') [ E' / E ] = (B [ E' / E ]) or (B' [ E' / E ]) .

eq ! E1 [E' / E] = ! (E1 [E' / E]) .
eq ? x : S [E' / E] = ? x : S .
eq ! E1 O [E' / E] = ! (E1 [E' / E]) (O[E' / E]) .
eq ? x : S O [E' / E] = ? x : S (O[E' / E]) .

eq A [E' / E] = A .
eq g'' O [E' / E] = g''(O[E' / E]) .
eq A:Action [SP] [E' / E] = (A:Action[E' / E]) [SP[E' / E]] .

eq (E1, DEL) [E' / E] = (E1[E' / E]), (DEL[E' / E]) .
eq nilDEL [E' / E] = nilDEL .

eq stop [E' / E] = stop .
eq exit [E' / E] = exit .
eq exit(E1) [E' / E] = exit(E1[E' / E]) .
eq exit(any S) [E' / E] = exit(any S) .
eq A:Action ; P [E' / E] = (A:Action[E' / E]) ; (P[E' / E]) .
eq P1 [] P2 [E' / E] = (P1[E' / E]) [] (P2[E' / E]) .
eq choice x : S [] P [E' / E] =
   choice x : S [] if x =/= E then (P [E' / E]) else P fi .
eq choice g in [GIL][] P [E' / E] = choice g in [GIL][] (P[E' / E]) .
eq P1 OP P2 [E' / E] = (P1[E' / E]) OP (P2[E' / E]) .
eq par g in [GIL] OP P [E' / E] = par g in [GIL] OP (P[E' / E]) .
eq P1 >> P2 [E' / E] = (P1[E' / E]) >> (P2[E' / E]) .
eq P1 >> accept x : S in P2 [E' / E] = (P1[E' / E]) >> accept x : S in
   if x =/= E then P2[E' / E] else P2 fi .
eq P1 [> P2 [E' / E] = (P1[E' / E]) [> (P2[E' / E]) .
eq [SP] -> P [E' / E] = [SP[E' / E]] -> (P[E' / E]) .
eq hide GIL in P [E' / E] = hide GIL in (P[E' / E]) .
eq let x = E in P [E' / E] = let x = E in if x =/= E then P[E' / E] else P fi .
eq p[GIL](DEL) [E' / E] = p[GIL](DEL[E' / E]) .
eq (E1 = E2) [ E' / E ] = (E1[ E' / E ]) = (E2[ E' / E ]) .
eq (b /\ b') [ E' / E ] = (b[ E' / E ]) /\ (b'[ E' / E ]) .

eq (g EO) [ E' / E ] = g (EO[ E' / E ]) .
eq (E1 EO) [ E' / E ] = (E1[ E' / E ]) (EO[ E' / E ]) .

*** gate substitution
eq i [g' / g] = i .
eq g'' [g' / g] = if g == g'' then g' else g'' fi .
eq g'' O [g' / g] = (g''[g' / g]) O .
eq A:Action [SP] [g' / g] = (A:Action[g' / g]) [SP] .

eq (g'', GIL) [g' / g] = (g''[g' / g]), (GIL[g' / g]) .

eq stop [g' / g] = stop .
eq exit [g' / g] = exit .
eq exit(E) [g' / g] = exit(E) .
eq exit(any S) [g' / g] = exit(any S) .
eq A:Action ; P [g' / g] = (A:Action[g' / g]) ; (P[g' / g]) .
eq P1 [] P2 [g' / g] = (P1[g' / g]) [] (P2[g' / g]) .
eq choice x : S [] P [g' / g] = choice x : S [] (P [g' / g]) .
eq choice g'' in [GIL][] P [g' / g] =
    if g == g'' then choice g'' in [GIL][] P
    else choice g'' in [GIL[g' / g]][] (P[g' / g]) fi .
eq P1 |[GIL]| P2 [g' / g] = (P1[g' / g]) |[(GIL[g' / g])]| (P2[g' / g]) .
eq P1 || P2 [g' / g] = (P1[g' / g]) || (P2[g' / g]) .
eq P1 ||| P2 [g' / g] = (P1[g' / g]) ||| (P2[g' / g]) .
eq par g'' in [GIL]|[GIL']| P [g' / g] =
    if g == g'' then par g'' in [GIL]|[GIL']| P
    else par g'' in [GIL[g' / g]]|[GIL'[g' / g]]| (P[g' / g]) fi .
eq par g'' in [GIL] || P [g' / g] =
    if g == g'' then par g'' in [GIL]|| P
    else par g'' in [GIL[g' / g]] || (P[g' / g]) fi .
eq par g'' in [GIL] ||| P [g' / g] =
    if g == g'' then par g'' in [GIL]||| P
    else par g'' in [GIL[g' / g]] ||| (P[g' / g]) fi .
eq P1 >> P2 [g' / g] = (P1[g' / g]) >> (P2[g' / g]) .
eq P1 >> accept x : S in P2 [g' / g] = (P1[g' / g]) >> accept x : S in (P2[g' / g]) .
eq P1 [> P2 [g' / g] = (P1[g' / g]) [> (P2[g' / g]) .
eq [SP] -> P [g' / g] = [SP] -> (P [g' / g]) .
eq hide GIL in P [g' / g] = hide (GIL[g' / g]) in (P[g' / g]) .
eq let x = E in P [g' / g] = let x = E in (P[g' / g]) .
eq p[GIL](DEL) [g' / g] = p[ GIL[g' / g] ](DEL) .
```

The function **name** : Act $\cup$ $\{\delta, \mathbf{i}\} \rightarrow G$ $\cup$ $\{\delta, \mathbf{i}\}$ extracts the gate name from a structured event, and is defined in [16].

```
op name : Event -> Event .

eq name(i) = i .
eq name(g) = g .
eq name(g EO) = g .
```

A predicate to know if a given gate identifier appears in a given gate identifier list is also used by the semantics.

```
op _in_ : Event GateIdList -> Bool .

eq i in GIL = false .
eq g in nilGIL = false .
eq g in ALL = true .
eq g in g' = (g == g') .
eq g in (g' , GIL) = (g == g') or (g in GIL) .
```

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 4.1 how it is defined automatically for new data expressions.

```
sort VarSet .
subsort VarId < VarSet .

op mt : -> VarSet .
op _U_ : VarSet VarSet -> VarSet [assoc comm id: mt] .

eq x U x = x . *** idempotency

op _in_ : VarId VarSet -> Bool .
op _subseteq_ : VarSet VarSet -> Bool .
op _\_ : VarSet VarSet -> VarSet .

vars VS VS' : VarSet .

eq x in mt = false .
eq x in (y U VS) = (x == y) or (x in VS) .

eq mt subseteq VS = true .
eq (x U VS) subseteq VS' = (x in VS') and (VS subseteq VS') .

eq mt \ VS' = mt .
eq (y U VS) \ VS' = if (y in VS') then VS \ VS'
                    else y U (VS \ VS') fi .

op vars : BehaviourExp -> VarSet .
op vars : DataExp -> VarSet .
op vars : DataExpList -> VarSet .
op vars : Offer -> VarSet .

eq vars(! E) = vars(E) .
eq vars(? x : S) = x .
eq vars(! E O) = vars(E) U vars(O) .
eq vars(? x : S O) = x U vars(O) .

eq vars(stop) = mt .
eq vars(exit) = mt .
eq vars(exit(E)) = vars(E) .
eq vars(exit(any S)) = mt .
eq vars(A ; P) = vars(P) .
eq vars(g O ; P) = vars(O) U vars(P) .
eq vars(g O [ SP ] ; P) = vars(O) U vars(SP) U vars(P) .
eq vars(P1 [] P2) = vars(P1) U vars(P2) .
eq vars(choice x : S [] P) = x U vars(P) .
eq vars(choice g in [GIL] [] P) = vars(P) .
eq vars(P1 OP P2) = vars(P1) U vars(P2) .
eq vars(par g in [GIL] OP P) = vars(P) .
eq vars(P1 >> P2) = vars(P1) U vars(P2) .
eq vars(P1 >> accept x : S in P2) = x U vars(P1) U vars(P2) .
eq vars(P1 [> P2) = vars(P1) U vars(P2) .
eq vars([SP] -> P) = vars(SP) U vars(P) .
eq vars(hide GIL in P) = vars(P) .
eq vars(let x = E in P) = x U vars(E) U vars(P) .
eq vars(p[GIL](DEL)) = vars(DEL) .

eq vars(x) = x .
eq vars(true) = mt .
eq vars(false) = mt .
eq vars(not(B)) = vars(B) .
eq vars(B and B') = vars(B) U vars(B') .
eq vars(B or B') = vars(B) U vars(B') .

eq vars(nilDEL) = mt .
eq vars((E, DEL)) = vars(E) U vars(DEL) .
```

The set of free variables occurring in a data or behaviour expression is defined as follows.

```
op fv : BehaviourExp -> VarSet .
op fv : DataExp -> VarSet .
op fv : DataExpList -> VarSet .
op fv : Offer -> VarSet .
op bv : Offer -> VarSet .

eq fv(E) = vars(E) .

eq fv(stop) = mt .
eq fv(exit) = mt .
eq fv(exit(E)) = fv(E) .
eq fv(exit(any S)) = mt .
eq fv(A ; P) = fv(P) .
eq fv(g O ; P) = fv(O) U (fv(P) \ bv(O)) .
eq fv(g O [ SP ] ; P) = fv(O) U ((fv(SP) U fv(P)) \ bv(O)) .
eq fv(P1 [] P2) = fv(P1) U fv(P2) .
eq fv(choice x : S [] P) = fv(P) \ x .
eq fv(choice g in [GIL] [] P) = fv(P) .
eq fv(P1 OP P2) = fv(P1) U fv(P2) .
eq fv(par g in [GIL] OP P) = fv(P) .
eq fv(P1 >> P2) = fv(P1) U fv(P2) .
eq fv(P1 >> accept x : S in P2) = fv(P1) U (fv(P2) \ x) .
eq fv(P1 [> P2) = fv(P1) U fv(P2) .
eq fv([SP] -> P) = fv(SP) U fv(P) .
eq fv(hide GIL in P) = fv(P) .
eq fv(let x = E in P) = fv(E) U (fv(P) \ x) .
eq fv(p[GIL](DEL)) = fv(DEL) .

eq fv(nilDEL) = mt .
eq fv((E, DEL)) = fv(E) U fv(DEL) .
```

```
  eq fv(! E) = fv(E) .
  eq fv(? x : S) = mt .
  eq fv(! E O) = fv(E) U fv(O) .
  eq fv(? x : S O) = mt U fv(O) .

  eq bv(! E) = mt .
  eq bv(? x : S) = x .
  eq bv(! E O) = bv(O) .
  eq bv(? x : S O) = x U bv(O) .
```

Now, we can represent the LOTOS symbolic semantics rules in Maude. For each LOTOS operator, we present the semantic rules and their representation as rewrite rules.

prefix axioms

$$a; P \xrightarrow{\text{tt} \qquad a} P$$

$$g\ d_1 \dots d_n; P \xrightarrow{\text{tt} \qquad gE'_1 \dots E'_n} P$$

$$g\ d_1 \dots d_n[SP]; P \xrightarrow{SP \qquad gE'_1 \dots E'_n} P$$

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

```
  *** prefix axioms
  rl [prefix] : A ; P => {true}{A}P .
  rl [prefix] : g O ; P => {true}{g eOffer(O)}P .
  rl [prefix] : g O [SP] ; P => {SP}{g eOffer(O)}P .

  op eOffer : Offer -> EOffer .

  eq eOffer(! E) = E .
  eq eOffer(? x : S) = x .
  eq eOffer(O O') = (eOffer(O) eOffer(O')) .
```

exit axioms

$$\textbf{exit} \xrightarrow{\text{tt} \qquad \delta} \textbf{stop}$$

$$\textbf{exit}(\text{ep}) \xrightarrow{\text{tt} \qquad \delta E'} \textbf{stop}$$

$$E' = \begin{cases} E & \text{if ep} = E \\ z & \text{if ep} = \textbf{any } S \quad \text{where } z \in \textbf{new-var}. \end{cases}$$

```
  *** exit axioms
  rl [exit] : exit => {true}{delta}stop .
  rl [exit] : exit(E) => {true}{delta E}stop .
  rl [exit] : exit(any S) => {true}{delta new-var}stop .
```

let rule

$$\frac{P[E/x] \xrightarrow{b \qquad \alpha} P'}{\textbf{let } x = E \textbf{ in } P \xrightarrow{b \qquad \alpha} P'}$$

```
  *** let rule
  crl [let] : let x = E in P => {b}{a}P' if P [E / x] => {b}{a}P' .
```

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

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\textbf{choice } x : S \,[\,]\, P \xrightarrow{b \quad \alpha} P'}$$

```
*** choice range rules
crl [choicer] : choice g in [GIL][] P => {b}{a}P'
 if select(GIL) => gi /\ P[gi / g] => {b}{a}P' .

crl [choicer] : choice x : S [] P => {b}{a}P' if P => {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) .
```

par rule

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

where *op* is one of the parallel operators, $\|$, $|||$, or $|[h_1, \ldots, h_m]|$,
for some gate names $h_1, \ldots, h_m$.

```
*** par rule
crl par g in [GIL] OP P => {b}{a}P'
 if unfold(g, GIL, OP, P) => {b}{a}P' .

op unfold : GateId GateIdList ParOp BehaviourExp -> BehaviourExp .

eq unfold(g, g', OP, P) = P[g' / g] .
eq unfold(g, (g', GIL), OP, P) = (P[g' / g]) OP unfold(g, GIL, OP, P) .
```

hide rules

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

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

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

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

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

accept rules

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \gg \textbf{accept } x\!:\!S \textbf{ in } P_2 \xrightarrow{b \quad \alpha} P_1' \gg \textbf{accept } x\!:\!S \textbf{ in } P_2}$$

if $\textbf{name}(\alpha) \neq \delta$

$$\frac{P_1 \xrightarrow{b \quad \delta E} P_1'}{P_1 \gg \textbf{accept } x\!:\!S \textbf{ in } P_2 \xrightarrow{b \quad i} P_2[E/x]}$$

Similarly for $\gg$ with no data.

```
*** accept rules
crl [accept] : P1 >> accept x : S in P2 => {b}{a}(P’ >> accept x : S in P2)
   if P1 => {b}{a}P’ /\ (name(a) =/= delta) .
crl [accept] : P1 >> accept x : S in P2 => {b}{i}(P2 [E / x])
   if P1 => {b}{delta E}P’ .
crl [accept] : P1 >> P2 => {b}{a}(P’ >> P2)
   if P1 => {b}{a}P’ /\ (name(a) =/= delta) .
crl [accept] : P1 >> P2 => {b}{i}P2
   if P1 => {b}{delta}P’ .
```

disable rules

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \,[\!> P_2 \xrightarrow{b \quad \alpha} P_1' \,[\!> P_2}$$

if $\textbf{name}(\alpha) \neq \delta$

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \,[\!> P_2 \xrightarrow{b \quad \alpha} P_1'}$$

if $\textbf{name}(\alpha) = \delta$

$$\frac{P_2 \xrightarrow{b \quad \alpha} P_2'}{P_1 \,[\!> P_2 \xrightarrow{b \quad \alpha} P_2'}$$

```
*** disable rules
crl [disable] : P1 [> P2 => {b}{a}(P’ [> P2)
   if P1 => {b}{a}P’ /\ (name(a) =/= delta) .
crl [disable] : P1 [> P2 => {b}{a}P’
   if P1 => {b}{a}P’ /\ (name(a) == delta) .
crl [disable] : P1 [> P2 => {b}{a}P’
   if P2 => {b}{a}P’ .
```

general parallelism rules (not synchronising)

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

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

$\sigma = \sigma_1 \dots \sigma_n$, $\alpha = gE_1 \dots E_n$, and

$$\sigma_i = \begin{cases} [z_i/x_i] & \text{if } E_i = x_i \text{ and } x_i \in \mathit{vars}(P_2) \\ [\,] & \text{otherwise} \end{cases} \quad \text{where } z_i \in \textbf{new-var}.$$

Similarly for P_2.

```
*** general parallelism rules (not synchronising)
crl [genpar] : P1 |[ GIL ]| P2 => {b subsPar(a, vars(P2))}
                      {a subsPar(a, vars(P2))}
                      ((P' subsPar(a, vars(P2))) |[ GIL ]| P2)
 if P1 => {b}{a}P' /\ not(name(a) in (GIL, delta)) .
crl [genpar] : P1 |[ GIL ]| P2 => {b subsPar(a, vars(P1))}
                      {a subsPar(a, vars(P1))}
                      (P1 |[ GIL ]| (P' subsPar(a, vars(P1))))
 if P2 => {b}{a}P' /\ not(name(a) in (GIL, delta)) .

eq P1 ||| P2 = P1 |[ nilGIL ]| P2 .
```

The following operation returns the substitution needed in the semantic rules for the parallel operator, depending on the given action.

```
op subsPar : Event VarSet -> Substitution .
op subsPar : EOffer VarSet -> Substitution .

eq subsPar(A, VS) = [] .
eq subsPar(g EO, VS) = subsPar(EO, VS) .

eq subsPar(E, VS) = if (E :: VarId) then
                      (if (E in VS) then [ new-var / E ] else [] fi)
                   else [] fi .
eq subsPar(E EO, VS) = if (E :: VarId) then
                         (if (E in VS) then [ new-var / E ] subsPar(EO, VS)
                          else subsPar(EO, VS) fi)
                      else subsPar(EO, VS) fi .
```

general parallelism rules (synchronising)

$$\frac{P_1 \xrightarrow{b_1 \qquad g} P_1' \qquad\qquad P_2 \xrightarrow{b_2 \qquad g} P_2'}{P_1 |[g_1, \dots, g_n]| P_2 \xrightarrow{b_1 \wedge b_2 \qquad g} P_1' |[g_1, \dots, g_n]| P_2'}$$

where $g \in \{g_1, \dots, g_n, \delta\}$

$$\frac{P_1 \xrightarrow{b_1 \qquad gE_1 \dots E_n} P_1' \qquad\qquad P_2 \xrightarrow{b_2 \qquad gE_1' \dots E_n'} P_2'}{P_1 |[g_1, \dots, g_n]| P_2 \xrightarrow{b_1 \wedge b_2 \wedge E_1 = E_1' \wedge \dots \wedge E_n = E_n' \qquad gE_1 \dots E_n} P_1' |[g_1, \dots, g_n]| P_2'}$$

where $g \in \{g_1, \dots, g_n, \delta\}$, and
when $vars(b_1 \cup E_1 \dots E_n) \cap vars(b_2 \cup E_1' \dots E_n') = \emptyset$.

```
*** general parallelism rules (synchronising without values)
crl [genpar] : P1 |[ GIL ]| P2 => {b1 /\ b2}{g}(P1' |[ GIL ]| P2')
 if P1 => {b1}{g}P1' /\ P2 => {b2}{g}P2' /\ (g in (GIL, delta)) .

*** general parallelism rules (synchronising with values)
crl [genpar] : P1 |[ GIL ]| P2 => {b1 /\ b2 /\ match(EO1,EO2)}
                                  {g EO1}(P1' |[ GIL ]| P2')
 if P1 => {b1}{g EO1}P1' /\
    P2 => {b2}{g EO2}P2' /\ (g in (GIL, delta)) .

eq P1 || P2 = P1 |[ ALL ]| P2 .

op match : EOffer EOffer -> TransCond .
eq match(E1, E2) = (E1 = E2) .
eq match(E1 EO1, E2 EO2) = (E1 = E2) /\ match(EO1, EO2) .
```

choice rules

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \; [\,] \; P_2 \xrightarrow{b \quad \alpha} P_1'}$$

$$\frac{P_2 \xrightarrow{b \quad \alpha} P_2'}{P_1 \; [\,] \; P_2 \xrightarrow{b \quad \alpha} P_2'}$$

```
*** choice rules
crl [choice] : P1 [] P2 => {b}{a}P' if P1 => {b}{a}P' .
crl [choice] : P1 [] P2 => {b}{a}P' if P2 => {b}{a}P' .
```

guard rule

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

```
*** guard rule
crl [guard] : [ SP ] -> P => {b /\ SP}{a}P' if P => {b}{a}P' .
```

stop rule

stop generates no rules.

instantiation rule

$$\frac{P[g_1/h_1, \ldots, g_n/h_n][E_1/x_1, \ldots, E_m/x_m] \xrightarrow{b \quad \alpha} P'}{p[g_1, \ldots, g_n](E_1, \ldots, E_m) \xrightarrow{b \quad \alpha} P'}$$

where $p[h_1, \ldots, h_n](x_1, \ldots, x_m) := P$ is a process definition

```
*** instantiation rule
crl [instantiation] : p[GIL](DEL) => {b}{a}P' if (p definedIn context) /\
 instantiate(def(context,p),GIL,DEL) => {b}{a}P' .
```

The operation `instantiate`, given a process definition and the gate identifiers and data expressions used in a process instantiation, returns the behaviour that defines the process, where formal gate identifiers and data parameters have been substituted by the actual ones.

```
op instantiate : ProcDef GateIdList DataExpList -> BehaviourExp .
op instantiate : BehaviourExp GateIdList GateIdList -> BehaviourExp .
op instantiate : BehaviourExp DataExpList DataExpList -> BehaviourExp .

eq instantiate(process(p,GIL,DEL,P), GIL', DEL') =
    instantiate(instantiate(P,GIL,GIL'), DEL, DEL') .

eq instantiate(P, nilGIL, nilGIL) = P .
eq instantiate(P, g, g') = P [ g' / g ] .
eq instantiate(P, (g, GIL), (g', GIL')) = instantiate(P,GIL,GIL') [ g' / g ] .

eq instantiate(P, nilDEL, nilDEL) = P .
eq instantiate(P, E, E') = P [ E' / E ] .
eq instantiate(P, (E, DEL), (E', DEL')) = instantiate(P,DEL,DEL') [ E' / E ] .
```

Now we have implemented all the semantics rules for behaviour expressions, and we have the following conservativity result.

Theorem 1 (Conservativity of LOTOS transitions).
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.

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.

```
ceq (E1 = E2) = (E2 = E1) if E2 :: VarId and not(E1 :: VarId) .

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)) .
```

In [4], 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 implement these transitions as we have done for behaviour expressions. The term T_σ is represented as the pair `<`T`,`σ`>`.

```
sorts term substitution .

op <_`,_> : BehaviourExp substitution -> term [frozen] .

op `(`) : -> substitution .
op `(_<-_`) : DataExp VarId -> substitution .
op __ : substitution substitution -> substitution [assoc id: ()] .
op _`[_/_`] : substitution DataExp VarId -> substitution .
op remove : substitution VarId -> substitution .

var sig : substitution .

eq sig [ E / x ] = remove(sig,x) (E <- x) .

eq remove((), x) = () .
eq remove((E <- y) sig, x) = if (x == y) then sig
                             else (E <- y) remove(sig, x) fi .

op __ : TransCond substitution -> TransCond .
op __ : Event substitution -> Event .

eq b () = b .
eq b ((E <- x) sig) = b [E / x] sig .
```

```
  eq a () = a .
  eq a ((E <- x) sig) = a [E / x] sig .

  op _<|_ : VarSet substitution -> substitution .

  eq VS <| () = () .
  eq VS <| ((E <- x) sig) = if (x in VS) then (E <- x) (VS <| sig)
                            else (VS <| sig) fi .

  sort TCondEventTerm .
  subsort term < TCondEventTerm .
  op {_}{_}_ : TransCond Event TCondEventTerm -> TCondEventTerm [frozen] .
```

Transitions between terms are defined in [4] as:

$$
\begin{array}{rll}
T \xrightarrow{b \quad a} T' & \text{implies} & T_\sigma \xrightarrow{b\sigma \quad a} T'_{\sigma'} \\
T \xrightarrow{b \quad gE} T' & \text{implies} & T_\sigma \xrightarrow{b\sigma \quad gE\sigma} T'_{\sigma'} \\
 & & \text{where } fv(E) \subseteq fv(T) \\
T \xrightarrow{b \quad gx} T' & \text{implies} & T_\sigma \xrightarrow{b\sigma[z/x] \quad gz} T'_{\sigma'[z/x]} \\
 & & \text{where } x \notin fv(T) \text{ and } z \notin fv(T_\sigma)
\end{array}
$$

In all cases, $\sigma' = fv(T') \lhd \sigma$. (By $s \lhd \sigma$ we mean domain restriction as in the Z notation, ie., the restriction of σ to include only domain elements in the set s.)

```
  crl [term] : < P, sig > => {b}{A} < P', fv(P') <| sig >
   if P => {b}{A}P' .
  crl [term] : < P, sig > => {b sig}{(g E) sig} < P', fv(P') <| sig >
   if  P => {b}{g E}P' /\ fv(E) subseteq fv(P) .
  crl [term] : < P, sig > => {b (sig [ z / x ])}{g z} < P', (fv(P') <| sig) [ z / x ] >
   if  P => {b}{g x}P' /\ z := new-var /\ not(x in fv(P)) .

endm
```

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.

3 FULL modal logic

A modal logic FULL has been described in [2] based on the LOTOS symbolic semantics.

We have implemented a subset of FULL without data values. 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 [5].

The FULL semantics uses the *successors* of a behaviour expression after performing an action. We define first operations to obtain all the possible transitions of a behaviour expression, that will be also used by our tool in order to execute or *simulate* processes. This operations are defined at the metalevel, and use the module with the LOTOS semantics rules.

```
fmod SUCC is
  pr META-LEVEL .

  sort TermSeq .
  subsort Term < TermSeq .
  op mt : -> TermSeq .
  op _+_ : TermSeq TermSeq -> TermSeq [assoc id: mt] .

  var M : Module .
  vars T T' T'' T1 T2 T3 : Term .
  var TS : TermSeq .
  var N : MachineInt .
```

The operation `transitions` 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 it 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.

```
  op transitions : Module Term -> TermSeq .
  op transitions : Module Term MachineInt -> TermSeq .

  eq transitions(M, T) = transitions(M,T,0) .

  eq transitions(M, T, N) =
     if metaSearch(M, T, 'X:TCondEventBehExp, nil, '+, 1, N) == failure
     then mt
     else getTerm(metaSearch(M, T, 'X:TCondEventBehExp, nil, '+, 1, N))
```

```
          + transitions(M, T, N + 1) fi .

  op transitions-subst : Module Term -> TermSeq .
  op apply-subst : Module TermSeq -> TermSeq .

  eq transitions-subst(M, T) = apply-subst(M, transitions(M, T, 0)) .

  eq apply-subst(M, mt) = mt .
  eq apply-subst(M, ('`{_`}`{_`}_[T1,T2,T3]) + TS) =
     getTerm(metaReduce(M, '`{_`}`{_`}_[T1,T2,'apply-subst[T1,T3]])) +
     apply-subst(M, TS) .
```

The operation `ttranstions` is the corrresponding version for term transitions.

```
  op ttransitions : Module Term -> TermSeq .
  op ttransitions : Module Term MachineInt -> TermSeq .

  eq ttransitions(M, T) = ttransitions(M,T,0) .

  eq ttransitions(M, T, N) =
     if metaSearch(M, T, 'X:TCondEventTerm, nil, '+, 1, N) == failure
     then mt
     else getTerm(metaSearch(M, T, 'X:TCondEventTerm, nil, '+, 1, N))
          + ttransitions(M, T, N + 1) fi .
```

We can define now an operation `succ` that returns all the successors of a behaviour expression after performing a given action. Buy now, we use the (metarepresented) module `LOTOS-SEMANTICS` without extensions because we do not allow data values in the modal logic operators.

```
  op succ : Term -> TermSeq .
  op succ : Term Term Term -> TermSeq .
  op filter : TermSeq Term Term -> TermSeq .
  op equal : Term Term -> Bool .


  eq succ(T) = transitions(['LOTOS-SEMANTICS], T, 0) .
  eq succ(T,T1,T2) = filter(succ(T),T1,T2) .

  eq filter(mt, T1, T2) = mt .
  eq filter(('`{_`}`{_`}_[T,T',T'']) + TS, T1, T2) =
     if equal(T,T1) and equal(T',T2) then
       T'' + filter(TS,T1,T2)
     else
       filter(TS,T1,T2)
     fi .

***  eq equal(T, T') = getTerm(metaReduce(['LOTOS-SEMANTICS], '_==_[T,T'])) == 'true.Bool .
   eq equal(T, T') = (T == T') .

endfm
```

The following module contains the representation of the subset of the FULL logic we implement. It presents the syntax of FULL and its semantics. The term `P |= Phi` is a Boolean, and it is `true` if the behaviour expresssion `P` satisfies the FULL formula `Phi`. Notice the use of the operations `forall` and `exists` for representing the universal and existential quantifiers.

```
mod FULL is
  protecting LOTOS-SYNTAX .
  protecting SUCC .
  protecting MOVE-UP .
  sort Formula .

  ops tt ff : -> Formula .
  ops _/\_  _\/_ : Formula Formula -> Formula .
  op <_>_ : SimpleAction Formula -> Formula .
  op [_]_ : SimpleAction Formula -> Formula .

  op forall : TermSeq Formula -> Bool .
  op exists : TermSeq Formula -> Bool .

***  sort Judgement .
  op _|=_ : BehaviourExp Formula -> Bool .
  op _|=_ : Term Formula -> Bool .

  var P : BehaviourExp .
  var A : SimpleAction .
  var T : Term .
  var TS : TermSeq .
  vars Phi Psi : Formula .

  eq P |= Phi = up(P) |= Phi .

  eq T |= tt = true .
  eq T |= ff = false .

  eq T |= Phi /\ Psi = (T |= Phi) and (T |= Psi) .

  eq T |= Phi \/ Psi = T |= Phi or  T |= Psi .

  eq T |= [ A ] Phi = forall(succ(T, up(true), up(A)), Phi) .

  eq T |= < A > Phi = exists(succ(T, up(true), up(A)), Phi) .

  eq forall(mt, Phi) = true .
  eq forall(T + TS, Phi) = (T |= Phi) and forall(TS, Phi) .

  eq exists(mt, Phi) = false .
  eq exists(T + TS, Phi) = T |= Phi or exists(TS,Phi) .

endm
```

4 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 [6]. This is specially important when $\mathcal{L}$ has user-definable syntax, as it is our case with ACT ONE. The grammar of ACT ONE is given in Appendix A.

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.

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

With our translation we achieve the following result.

Theorem 2 (Conservativity of ACT ONE equivalence).
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 $M =$ `translate`(`metaParse`(`LOTOS-GRAMMAR`,SP)), and t_M and t'_M are the representations of t and t' in M.

```
fmod ACTONE-TRANSLATION is
  pr AUX-FUNC .
  pr UNIT .

  sort VarSet .
  op _:_ : Qid Qid -> VarSet .
  op mt : -> VarSet .
  op __ : VarSet VarSet -> VarSet [assoc comm id: mt] .

  op extractSignature : Term -> FModule .
  op extractSignature : Term FModule -> FModule .
  op extractVariables : Term -> VarSet .
  op insertVariables : QidList VarSet -> QidList .
  op insertVariablesAux : QidList VarSet -> QidList .
  op createVariables : QidList Qid -> VarSet .
  op listToQidList : Term -> QidList .
  op translateType : Term -> FModule .
  op translateType : Term FModule FModule -> FModule .
  op translateDeclList : Term FModule FModule -> FModule .
  op translateOpDeclList : Term FModule FModule -> FModule .
  op translateVarEqDeclList : Term FModule FModule VarSet -> FModule .
  op translateEqDeclList : Term FModule FModule VarSet -> FModule .

  vars Q QI V S : Qid .
  vars T T' T'' T''' : Term .
  vars M M' : Module .
  var VS : VarSet .
  var QIL : QidList .

  eq extractSignature('type_is_endtype['token[T'],T'']) =
     extractSignature(T'',
         addImportList(including 'BOOL .,emptyFModule(downQid(T')))) .
  eq extractSignature('type_is__endtype['token[T],T', T'']) =
     extractSignature('type_is_endtype['token[T],T'']) .
  eq extractSignature('__[T,T'], M) =
     extractSignature(T', extractSignature(T,M)) .
  eq extractSignature('sorts_['token[T]], M) =
     addSortSet(downQid(T), M) .
  eq extractSignature('eqns_[T], M) = M .
  eq extractSignature('opns_[T], M) = extractSignature(T,M) .
  eq extractSignature('_:`->_[T,'token[T'']],M) =
     addOps(listToQidList(T), nil, downQid(T''), M) .
  eq extractSignature('_:_->_[T,T'','token[T''']], M) =
     addOps(listToQidList(T), listToQidList(T''), downQid(T'''), M) .

  eq listToQidList('token[T]) = downQid(T) .
  eq listToQidList('_`,_[T,T']) =
     listToQidList(T) listToQidList(T') .

  eq translateType(T) = translateType(T, addImportList(including
     'DATAEXP-SYNTAX ., emptyFModule), emptyFModule) .

  eq translateType('__[T,T'], M, M') =
      translateType(T', translateType(T, M, M'),
                        addDecls(M', extractSignature(T))) .

  eq translateType('type_is_endtype['token[T'],T''], M, M') =
     translateDeclList(T'', M,
       addDecls(M',extractSignature('type_is_endtype['token[T'],T'']))) .
  eq translateType('type_is__endtype['token[T],T', T''], M, M') =
     translateType('type_is_endtype['token[T],T''], M, M') .
  eq translateDeclList('__[T,T'], M, M') =
     translateDeclList(T', translateDeclList(T,M, M'), M') .
```

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 values 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.

```
  eq translateDeclList('sorts_['token[T]], M, M') =
     addSubsortDeclSet(subsort downQid(T) < 'DataExp .,
       addSubsortDeclSet(subsort 'VarId < downQid(T) .,
         addSortSet(downQid(T), M))) .
  eq translateDeclList('opns_[T], M, M') = translateOpDeclList(T, M, M') .
  eq translateOpDeclList('__[T,T'], M, M') =
     translateOpDeclList(T', translateOpDeclList(T,M, M'), M') .
  eq translateOpDeclList('_:`->_[T,'token[T'']], M, M') =
     addOps(listToQidList(T), nil, downQid(T''), M) .
  eq translateOpDeclList('_:_->_[T,T'','token[T''']], M, M') =
     addOps(listToQidList(T), listToQidList(T''), downQid(T'''), M) .

  eq translateDeclList('eqns_[T], M, M') =
     translateVarEqDeclList(T, M, M', extractVariables(T)) .
  eq translateVarEqDeclList('__[T,T'], M, M', VS) =
     translateVarEqDeclList(T', translateVarEqDeclList(T, M, M', VS), M', VS) .
  eq translateVarEqDeclList('forall_[T], M, M', VS) = M .
  eq translateVarEqDeclList('ofsort__[T,T'], M, M', VS) =
     translateEqDeclList(T', M, M', VS) .

  eq translateEqDeclList('__[T,T'], M, M', VS) =
     translateEqDeclList(T', translateEqDeclList(T, M, M', VS), M', VS) .
  eq translateEqDeclList('_=_;['bubble[T], 'bubble[T']], M, M', VS) =
     addEquationSet(
           (eq getTerm(metaParse(M', insertVariables(downQidList(T), VS), anyType))
             = getTerm(metaParse(M', insertVariables(downQidList(T'), VS), anyType)) .),
          M) .
  eq translateEqDeclList('_=>_=_;['bubble[T], 'bubble[T'], 'bubble[T'']], M, M', VS) =
     addEquationSet(
           (ceq getTerm(metaParse(M', insertVariables(downQidList(T'), VS), anyType))
             = getTerm(metaParse(M', insertVariables(downQidList(T''), VS), anyType))
             if getTerm(metaParse(M', insertVariables(downQidList(T), VS), anyType))
                = 'true.Bool .),
          M) .

  eq extractVariables('__[T,T']) =
     extractVariables(T) extractVariables(T') .
  eq extractVariables('forall_[T]) = extractVariables(T) .
  eq extractVariables('ofsort__[T,T']) = mt .
  eq extractVariables('_`,_[T,T']) =
     extractVariables(T) extractVariables(T') .
  eq extractVariables('_:_[T,'token[T']]) =
     createVariables(listToQidList(T), downQid(T')) .

  eq createVariables(nil,S) = mt .
  eq createVariables(QI QIL, S) = (QI : S) createVariables(QIL, S) .

   eq insertVariables(QIL, mt) = QIL .
   eq insertVariables(QIL, (V : S) VS) =
      insertVariables(insertVariablesAux(QIL, V : S), VS) .

   eq insertVariablesAux(nil, V : S) = nil .
   eq insertVariablesAux(Q QIL, V : S) =
      if Q == V then
        conc(V, conc(':,S)) insertVariablesAux(QIL, V : S)
      else
        Q insertVariablesAux(QIL, V : S) fi .
endfm
```

The following is an example of how an ACT ONE specification is translated into a functional module. The ACT ONE specification on the left is translated into the functional module on the right.

```
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
```

4.1 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. In the module `MODULE-EXTENSIONS` below, 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) .
```

```
fmod MODULE-EXTENSIONS is
 pr UNIT .

 op addOpervars : Module -> Module .
 op addOpervars : OpDeclSet Module -> Module .
 op addOpervars : Qid TypeList Qid Module -> Module .
 op buildArgs : TypeList MachineInt -> TermList .
 op buildArgs2 : Qid TermList -> TermList .

 var M : Module .
 vars OP S A A' : Qid .
 var ARGS : TypeList .
 var T : Term .
 var TL : TermList .
 var AttS : AttrSet .
 var ODS : OpDeclSet .
 var N : MachineInt .

 eq addOpervars(M) = addOpervars(opDeclSet(M), M) .
```

[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.

```
 eq addOpervars(none, M) = M .
 eq addOpervars(op OP : ARGS -> S [AttS] . ODS, M) =
      addOpervars(ODS, addOpervars(OP, ARGS, S, M)) .

 eq addOpervars(OP, nil, S, M) =
      addEquationSet(eq 'vars[conc(OP,conc('.,S))] = 'mt.VarSet ., M) .
 eq addOpervars(OP, A ARGS, S, M) =
      addEquationSet(eq 'vars[OP[buildArgs(A ARGS, 1)]] =
         if ARGS == nil then 'vars[buildArgs(A ARGS, 1)]
         else '_U_[buildArgs2('vars, buildArgs(A ARGS, 1))] fi ., M) .

 eq buildArgs(A, N) = conc(conc(index('v,N),':), A) .
 eq buildArgs(A A' ARGS, N) = buildArgs(A, N), buildArgs(A' ARGS, N + 1) .

 eq buildArgs2(OP, T) = OP[T] .
 eq buildArgs2(OP, (T,TL)) = OP[T], buildArgs2(OP,TL) .

 op addOperSubs : Module -> Module .
 op addOperSubs : OpDeclSet Module -> Module .
 op addOperSubs : Qid TypeList Qid Module -> Module .
 op buildArgs3 : Qid TermList -> TermList .
 op subsDE : -> Term .

 eq subsDE = '‘[_/_‘]['E':DataExp,'E:DataExp] .

 eq addOperSubs(M) = addOperSubs(opDeclSet(M), M) .

 eq addOperSubs(none, M) = M .
 eq addOperSubs(op OP : ARGS -> S [AttS] . ODS, M) =
      addOperSubs(ODS, addOperSubs(OP, ARGS, S, M)) .

 eq addOperSubs(OP, nil, S, M) =
      addEquationSet(eq '__[conc(OP,conc('.,S)), subsDE]
                        = conc(OP,conc('.,S)) ., M) .
 eq addOperSubs(OP, A ARGS, S, M) =
      addEquationSet(eq '__[OP[buildArgs(A ARGS, 1)], subsDE] =
         OP[buildArgs3('__, buildArgs(A ARGS, 1))] ., M) .

 eq buildArgs3(OP, T) = OP[T, subsDE] .
 eq buildArgs3(OP, (T,TL)) = OP[T, subsDE], buildArgs3(OP,TL) .

endfm
```

5 Building the LOTOS tool

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 with the data types given in ACT ONE and the given process definitions.

5.1 The grammar of the LOTOS tool interface

We have to define first the signature (syntax) of LOTOS and the signature of the commands we are going to use in our tool to work with the entered specification.[3] The signature of ACT ONE and LOTOS is given in Appendix A.

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

```
fmod LOTOS-TOOL-SIGN is
  protecting LOTOS-SIGN .
  pr MACHINE-INT .

  sort LotosCommand .

  op show process . : -> LotosCommand .
  op show transitions . : -> LotosCommand .
  op show transitions of_. : BehaviourExp -> LotosCommand .
  op cont_. : MachineInt -> LotosCommand .
  op cont . :  -> LotosCommand .
  op show state . : -> LotosCommand .
  op show term transitions . : -> LotosCommand .
  op show term transitions of_. : BehaviourExp -> LotosCommand .
  op tcont_. : MachineInt -> LotosCommand .
  op tcont . :  -> 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. The last commands correspond to term transitions, instead of behaviour expressions transitions.

In order to parse some input using the built-in function `metaParse`, 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` we get the metarepresentation of the signature. The `LOTOS-GRAMMAR` module will be used in calls to the `metaParse` function in order to get the input parsed in this signature. Notice that from the call to `metaParse` 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.

```
fmod META-LOTOS-TOOL-SIGN is
  including META-LEVEL .

  op LOTOS-GRAMMAR : -> FModule .
  eq LOTOS-GRAMMAR
    = (fmod 'LOTOS-GRAMMAR is
        including 'QID-LIST .
        including 'LOTOS-TOOL-SIGN .
        sorts none .
```

[3]There is an important separation between the concrete signature used by the users to write their specifications and the abstract syntax we defined in Section 2.1, althouth apparently they are quite similar.

```
        none
        op 'token : 'Qid -> 'Token
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)))] .
        op 'neTokenList : 'QidList -> 'NeTokenList
             [special(
               (id-hook('Bubble, '1 '-1 '`( '`))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)
                id-hook('Exclude, '.)))] .
        op 'expBubble : 'QidList -> 'ExpBubble
             [special(
               (id-hook('Bubble, '1 '-1 '`( '`))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)
                id-hook('Exclude, '. '! '=> ';  'any 'ofsort '`[ '`] )))] .
        op 'bubble : 'QidList -> 'Bubble
             [special(
               (id-hook('Bubble, '1 '-1 '`( '`))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)
                id-hook('Exclude, '. '! '=> '; 'any 'ofsort '`[ '`] )))] .
        none
        none
     endfm) .

endfm
```

5.2 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 module `LOTOS-SYNTAX` (Section 2.1). The parse tree returned by `metaParse` 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 is obtained by translating the types defined in ACT ONE into functional modules, as explained above. 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 fact that `x` is a variable of sort `Nat`.

```
fmod LOTOS-PARSING is
  protecting META-LEVEL .
  pr ACTONE-TRANSLATION .

  vars QI F So Q Q1 Q2 : Qid .
  var QIL : QidList .
  vars N : MachineInt .
  var M : Module .
  vars T T' T'' T''' PS N' T1 T2 T3 T4 T5 E G X S : Term .
  var TL : TermList .
  var VS : VarSet .
  var V : Variable .
  var C : Constant .

  sort TermVars .
  op <_`,_> : Term VarSet -> TermVars .

  op term : TermVars -> Term .
  op vars : TermVars -> VarSet .
  eq term(< T, VS >) = T .
  eq vars(< T, VS >) = VS .
```

We use the operation `parseProcess` to make this translation. It receives as arguments the term returned by `metaParse` (representing a behaviour expression), the metarepresented module with the data types syntax (obtained from the ACT ONE specification) and the set of free variables that may appear in the behaviour expression. It returns a behaviour expression without bubbles. It uses the operation `parseAction` that, besides the term metarepresenting the given action (without bubbles), returns the variables declared in the action (if any).

```
  op parseProcess : Term Module VarSet -> Term .
  op parseExitParam : Term Module VarSet -> Term .
  op parseDataExp : Term Module VarSet -> Term .
  op parseAction : Term Module VarSet -> TermVars .
  op parseOffer : Term Module VarSet -> TermVars .
  op parseParOp : Term -> Term .
  op parseGateIdList : Term -> Term .
  op parseDataExpList : Term Module VarSet -> Term .
  op parseProcDeclList : Term Module -> Term .
  op parseVariDeclList : Term -> Term .
  op parseVariList : Term -> Term .
  op varsMaudeToLotos : TermList -> Term .
  op varDeclToVarSet : Term -> VarSet .
  op varDeclToVarSet : Term Qid -> VarSet .

  eq parseExitParam('expBubble[E], M, VS) =
         parseDataExp('expBubble[E], M, VS) .
  eq parseExitParam('any_['token[S]], M, VS) = 'any_['S[S]] .

  eq parseAction('token[G], M, VS) = < 'G[G], mt > .
  eq parseAction('i.SimpleAction, M, VS) = < 'i.SimpleAction, mt > .
  eq parseAction('__[G, T'], M, VS) =
      < '__[term(parseAction(G, M, VS)), term(parseOffer(T',M, VS))],
        vars(parseOffer(T',M, VS)) > .
  eq parseAction('_`[_`][T,T'], M, VS) =
      < '_`[_`][term(parseAction(T,M,VS)),
               parseDataExp(T',M,VS vars(parseAction(T,M,VS)))],
        vars(parseAction(T,M,VS)) > .

  eq parseOffer('__[T,T'], M, VS) =
     < '__[term(parseOffer(T, M, VS)), term(parseOffer(T', M, VS))],
       vars(parseOffer(T, M, VS)) vars(parseOffer(T', M, VS)) > .

  eq parseOffer('!_[E], M, VS) = < '!_[parseDataExp(E,M,VS)] , mt > .
  eq parseOffer('?_['_:_['token[X],'token[S]]], M, VS) =
      < '?_['_:_['V[X], 'S[S]]], downQid(X) : downQid(S) > .
```

```
eq parseProcess('stop.BehaviourExp, M, VS) = 'stop.BehaviourExp .
eq parseProcess('exit.BehaviourExp, M, VS) = 'exit.BehaviourExp .
eq parseProcess('exit`(_`)[T], M, VS) = 'exit`(_`)[parseExitParam(T,M,VS)] .
```

When parsing a process prefixed by an action, first the action is parsed and then the process is parsed using an extended set of variables that includes the variables declared by the action (if any).

```
eq parseProcess('_;_[T,T'],M,VS) =
    '_;_[term(parseAction(T,M,VS)),
         parseProcess(T',M, VS vars(parseAction(T,M,VS)))] .

eq parseProcess('_`[`]_[T,T'], M, VS) =
    '_`[`]_[parseProcess(T,M, VS), parseProcess(T',M, VS)] .
eq parseProcess('choice_`[`]_['_:_['token[T],'token[T']],T''], M, VS) =
    'choice_`[`]_['_:_['V[T], 'S[T']],
                 parseProcess(T'',M, VS (downQid(T) : downQid(T')))] .
eq parseProcess('choice_in`[_`]`[`]_['token[T],T',T''], M,VS) =
    'choice_in`[_`]`[`]_['G[T], parseGateIdList(T'), parseProcess(T'',M, VS)] .

eq parseProcess('___[T,T',T''], M, VS) =
    '___[parseProcess(T,M, VS), parseParOp(T'), parseProcess(T'',M, VS)] .
eq parseProcess('par_in`[_`]__['token[T],T',T'',T'''], M,VS) =
    'par_in`[_`]__['G[T], parseGateIdList(T'),
                        parseParOp(T''), parseProcess(T''',M, VS)] .

eq parseProcess('_>>_[T,T'], M,VS) =
    '_>>_[parseProcess(T,M, VS), parseProcess(T',M, VS)] .
eq parseProcess('_>>`accept_in_[T,'_:_['token[T'],'token[T'']],T'''], M,VS) =
    '_>>`accept_in_[parseProcess(T,M, VS), '_:_['V[T'], 'S[T'']],
                  parseProcess(T''',M, VS (downQid(T') : downQid(T'')))] .

eq parseProcess('_`[>_[T,T'], M,VS) =
    '_`[>_[parseProcess(T,M, VS), parseProcess(T',M, VS)] .

eq parseProcess('hide_in_[T,T'], M, VS) =
    'hide_in_[parseGateIdList(T), parseProcess(T',M, VS)] .

eq parseProcess('let_=_in_['token[T],T',T''], M, VS) =
    'let_=_in_['V[T],parseDataExp(T',M,VS),
              parseProcess(T'',M, VS (downQid(T) : 'VarId))] .

eq parseProcess('`[_`]->_[T,T'], M, VS) =
    '`[_`]->_[parseDataExp(T,M,VS), parseProcess(T',M,VS)] .

eq parseProcess('_`[_`]`(_`)['token[T],T',T''], M, VS) =
    '_`[_`]`(_`)['P[T],parseGateIdList(T'),parseDataExpList(T'', M, VS)] .
eq parseProcess('_`[`]`(_`)['token[T],T''], M, VS) =
    '_`[_`]`(_`)['P[T],'nilGIL.GateIdList,parseDataExpList(T'', M, VS)] .
eq parseProcess('_`[_`]['token[T],T'], M, VS) =
    '_`[_`]`(_`)['P[T],parseGateIdList(T'),'nilDEL.DataExpList] .
eq parseProcess('_`[`]['token[T]], M, VS) =
    '_`[_`]`(_`)['P[T],'nilGIL.GateIdList,'nilDEL.DataExpList] .

eq parseParOp('|`[_`]|[T]) = '|`[_`]|[parseGateIdList(T)] .
```

```
  eq parseParOp('||.ParOp) = '||.ParOp .
  eq parseParOp('|||.ParOp) = '|||.ParOp .

  eq parseGateIdList('token[G]) = 'G[G] .
  eq parseGateIdList('_`,_[T,T']) = '_`,_[parseGateIdList(T),
                                          parseGateIdList(T')] .
```

The operation `parseDataExp` receives an expression bubble, a module with the syntax with which the expression has to be parsed, and a set of LOTOS variables which may appear in the expression (that is, the expression was found in the *scope* of these variables). In order to correctly parse the expression bubble, information about the variables has to be included in the expression as Maude variables. The resulting term may have Maude variables, that have to be transformed in LOTOS variables (which have the form `V(Q)`, where `Q` is a quoted identifier).

```
  eq parseDataExp('expBubble[E], M, VS) =
     varsMaudeToLotos(getTerm(metaParse(M,
         insertVariables(downQidList(E),VS),anyType))) .

  eq parseDataExpList('expBubble[E], M, VS) = parseDataExp('expBubble[E], M, VS) .
  eq parseDataExpList('_`,_[T,T'], M, VS) =
      '_`,_[parseDataExpList(T, M, VS), parseDataExpList(T', M, VS)] .

  eq varsMaudeToLotos(V) = 'V[conc('',conc(getName(V),'.Qid))] .
  eq varsMaudeToLotos(C) = C .
  eq varsMaudeToLotos(F[TL]) = F[varsMaudeToLotos(TL)] .
  eq varsMaudeToLotos((T, TL)) =
     (varsMaudeToLotos(T), varsMaudeToLotos(TL)) .
```

The operation `parseProcDeclList` is used to build a metarepresented context that includes the definitions of the processes declared in a specification.

```
  eq parseProcDeclList('emptyProcList.ProcDeclList,M) = 'nil.Context .
  eq parseProcDeclList('process_`[_`]`(_`):_:=_endproc['token[T1],T2,T3,T4,T5],M) =
     'process['P[T1], parseGateIdList(T2), parseVariDeclList(T3),
              parseProcess(T5, M, varDeclToVarSet(T3)) ] .
  eq parseProcDeclList('process_`[_`]:_:=_endproc['token[T1],T2,T4,T5],M) =
     'process['P[T1], parseGateIdList(T2), 'nilDEL.DataExpList,
              parseProcess(T5, M, mt) ] .
  eq parseProcDeclList('__[T,T'], M) =
     '_&_[parseProcDeclList(T, M), parseProcDeclList(T', M)] .

  eq parseVariDeclList('_`,_[T,T']) =
      '_`,_[parseVariDeclList(T), parseVariDeclList(T')] .
  eq parseVariDeclList('_:_[T,T']) = parseVariList(T) .
  eq parseVariList('token[T]) = 'V[T] .
  eq parseVariList('_`,_[T,T']) =
      '_`,_[parseVariList(T), parseVariList(T')] .

  eq varDeclToVarSet('_`,_[T,T']) =
       varDeclToVarSet(T) varDeclToVarSet(T') .
  eq varDeclToVarSet('_:_[T,'token[T']]) = varDeclToVarSet(T,downQid(T')) .
  eq varDeclToVarSet('token[T], So) = downQid(T) : So .
  eq varDeclToVarSet('_`,_[T,T'], So) =
       varDeclToVarSet(T, So) varDeclToVarSet(T', So) .
  op parseFormula : Term -> Term .
  eq parseFormula(C) = C .
  eq parseFormula('_/\_[T,T']) = '_/\_[parseFormula(T),parseFormula(T')] .
  eq parseFormula('_\/_[T,T']) = '_\/_[parseFormula(T),parseFormula(T')] .
  eq parseFormula('`[_`]_['token[G],T']) = '`[_`]_['G[G], parseFormula(T')] .
  eq parseFormula('<_>_['token[G],T']) = '<_>_['G[G], parseFormula(T')] .

endfm
```

5.3 LOTOS command processing

The following module contains auxiliary functions that are used to process the tool commands. They will be used from the rules that define the tool state handling (see Section 5.4).

```
fmod LOTOS-COMMAND-PROCESSING is
  protecting SUCC .

  vars Q Q1 Q2 : Qid .
  var QIL : QidList .
  var M : Module .
  vars T T1 T2 T3 : Term .
  var TS : TermSeq .
  var N : MachineInt .

  op meta-pretty-print-transitions : Module TermSeq -> QidList .
  op meta-pretty-print-transitions : Module TermSeq MachineInt -> QidList .
  op meta-pretty-print-trace : Module Term -> QidList .
  op meta-pretty-print-condition : Module Term -> QidList .

  eq meta-pretty-print-transitions(M, TS) =
    if TS == mt then
       ('\n 'No 'more 'transitions '. '\n)
    else
      ('\n 'TRANSITIONS ': '\n
       meta-pretty-print-transitions(M, TS, 1) '\n )
    fi .

  eq meta-pretty-print-transitions(M, mt, N) = nil .
  eq meta-pretty-print-transitions(M, T + TS, N) =
    '\n conc(index(' , N), '.) '\t filter(metaPrettyPrint(M,T))
       meta-pretty-print-transitions(M, TS, N + 1) .

  eq meta-pretty-print-trace(M, T) =
       ('\n 'Trace ': filter(metaPrettyPrint(M,T)) '\n) .

  eq meta-pretty-print-condition(M, T) =
       ('\n 'Condition ': filter(metaPrettyPrint(M,T)) '\n) .

  op filter : QidList -> QidList .
  op filter2 : QidList -> QidList .

  eq filter(nil) = nil .
  eq filter(Q QIL) = if Q == 'V or Q == 'S or Q == 'G or Q == 'P then
                       filter2(QIL)
```

```
                    else (Q filter(QIL)) fi .
  eq filter2(Q1 Q Q2 QIL) = strip(Q) filter(QIL) .
```

The operation `selectSol` receives as arguments an integer `N` and a sequence of solutions returned by the operation `transitions`, and it returns the $\mathtt{N}^{th}$ element in the sequence.

```
  op selectSol : MachineInt TermSeq -> Term .
  op selectProc : Term -> Term .
  op selectEvent : Term -> Term .
  op selectCond : Term -> Term .

  eq selectSol(1, T + TS) = T .
  ceq selectSol(N, T + TS) = selectSol(_-_(N,1), TS) if N > 1 .

  eq selectProc('`{_`}`{_`}_[T1,T2,T3]) = T3 .
  eq selectEvent('`{_`}`{_`}_[T1,T2,T3]) = T2 .
  eq selectCond('`{_`}`{_`}_[T1,T2,T3]) = T1 .

endfm
```

5.4 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 [7]:

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

But, since we are using Core Maude, object-oriented declarations cannot be given directly. Instead, we use the equivalent declarations desugaring the desired object-oriented declarations, as explained in [12].

```
mod TOOL-STATE-HANDLING is
  pr CONFIGURATION .
  pr LOTOS-PARSING .
  pr LOTOS-COMMAND-PROCESSING .
  pr ACTONE-TRANSLATION .
  pr MODULE-EXTENSIONS .

  sort ToolStateClass .
  subsort ToolStateClass < Cid .
  op ToolState : -> ToolStateClass .

  *** Attributes of the tool state
  op input :_ : TermList -> Attribute .
  op output :_ : QidList -> Attribute .
  op dataExpSyntax :_ : Module -> Attribute .
  op semantics :_ : Module -> Attribute .
  op lotosProcess :_ : Term -> Attribute .
  op transitions :_ : TermSeq -> Attribute .
  op trace :_ : Term -> Attribute .
  op condition :_ : Term -> Attribute .

  op SYN : -> FModule .
  op SEM : -> Module .
  eq SYN = ['DATAEXP-SYNTAX] .
  eq SEM = ['LOTOS-SEMANTICS] .

  var TL : TermList .
  var Atts : AttributeSet .
  var X@ToolState : ToolStateClass .
  var O : Oid .
  vars T T' T'' T''' T1 T2 T3 T4 : Term .
  vars SynM SemM : Module .
  var TS : TermSeq .

  op nilTermList : -> TermList .
  eq (nilTermList, TL) = TL .
  eq (TL, nilTermList) = TL .
```

The following rules describes the behaviour of the tool when a LOTOS specification or the different commands are entered into the system.

The first rule 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 4.1); the resulting module is joined with the metarepresentation of module `LOTOS-SEMANTICS`; and, finally, an equation defining the constant `context` 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, and the rest of attributes are initialized. Notice the message placed in the output channel.

```
  rl [spec] :
     < O : X@ToolState |
        input : ('specification__behaviour_where_endspec['token[T],T',T'',T''']),
        output : nil,
        dataExpSyntax : SynM, semantics : SemM, lotosProcess : T1,
        transitions : TS, trace : T3, condition : T4, Atts >
```

```
    => < O : X@ToolState | input : nilTermList,
          output : ('\n 'Introduced 'specification getName(T) '\n),
          dataExpSyntax : addDecls(translateType(T'), SYN),
          semantics : addEquationSet(eq 'context.Context =
                            parseProcDeclList(T''',addDecls(translateType(T'), SYN)) .,
                        addDecls(SEM, addOperSubs(addOpervars(translateType(T'))))),
          lotosProcess : parseProcess(T'',addDecls(translateType(T'), SYN),mt),
          transitions : mt, trace : 'nil.Trace, condition : 'true.Bool, Atts > .

  *** when some part of the specification is missing
  rl [spec] :
    < O : X@ToolState |
      input : ('specification_behaviour_where_endspec[T,T'',T''']), Atts  >
    => < O : X@ToolState |
      input : ('specification__behaviour_where_endspec[T,
                'emptyTypeList.TypeDeclList, T'',T''']), Atts  > .

  rl [spec] :
    < O : X@ToolState |
      input : ('specification__behaviour_endspec[T,T',T'']), Atts  >
    => < O : X@ToolState |
      input : ('specification__behaviour_where_endspec[T,
                T', T'','emptyProcList.ProcDeclList]), Atts  > .

  rl [spec] :
    < O : X@ToolState |
      input : ('specification_behaviour_endspec[T,T'']), Atts  >
    => < O : X@ToolState |
      input : ('specification__behaviour_where_endspec[T,
               'emptyTypeList.TypeDeclList , T'',
               'emptyProcList.ProcDeclList]), Atts  > .
```

Commands are handled by rules as well. For example, the following rule handles the `show transitions` command.

```
  rl [show-transitions] :
    < O : X@ToolState |
      input : 'show`transitions`..LotosCommand,
      semantics : SemM,
      lotosProcess : T,
      transitions : TS, Atts >
    => < O : X@ToolState |
         input : 'show`state`..LotosCommand,
         semantics : SemM, lotosProcess : T,
         transitions : transitions-subst(SemM,T), Atts >  .

  rl [show-transitions] :
    < O : X@ToolState |
      input : 'trans`..LotosCommand,
      semantics : SemM,
      lotosProcess : T,
      transitions : TS, Atts >
    => < O : X@ToolState |
         input : 'show`state`..LotosCommand,
         semantics : SemM, lotosProcess : T,
          transitions : transitions-subst(SemM,T), Atts >  .

  rl [show-ttransitions] :
     < O : X@ToolState |
        input : 'ttrans`..LotosCommand,
        semantics : SemM,
        lotosProcess : T,
        transitions : TS, Atts >
     => < O : X@ToolState |
          input : 'show`state`..LotosCommand,
          semantics : SemM, lotosProcess : T,
          transitions : ttransitions(SemM,'<_`,_>[T,'`(`).substitution]), Atts >  .

  rl [show] :
     < O : X@ToolState |
        input : ('show`process`..LotosCommand),
        output : nil, semantics : SemM,
        lotosProcess : T, Atts >
     => < O : X@ToolState | input : nilTermList,
          output : ('This 'is 'a 'LOTOS 'process '. '\n
                    filter(metaPrettyPrint(SemM, T))),
          semantics : SemM, lotosProcess : T, Atts >  .

  rl [show] :
     < O : X@ToolState |
        input : ('show`context`..LotosCommand),
        output : nil, semantics : SemM, Atts >
     => < O : X@ToolState | input : nilTermList,
          output : ('LOTOS 'context:  '\n
                    metaPrettyPrint(SemM,
                    getTerm(metaReduce(SemM, 'context.Context)))),
          semantics : SemM, Atts >  .

  rl [continue] :
     < O : X@ToolState |
        input : 'cont`..LotosCommand, Atts >
     => < O : X@ToolState |
          input : ('cont_.['1.NzMachineInt]), Atts > .

  rl [continue] :
     < O : X@ToolState |
        input : ('cont_.[T]), output : nil,
        semantics : SemM,
        lotosProcess : T', transitions : TS, trace : T'', condition : T''', Atts >
     => < O : X@ToolState |
        input : 'trans`..LotosCommand, output : nil,
        semantics : SemM,
        lotosProcess : selectProc(selectSol(downMachineInt(T),TS)),
        transitions : TS,
        trace : getTerm(metaReduce(SemM,
                 '__[T'',selectEvent(selectSol(downMachineInt(T),TS))])),
        condition : getTerm(metaReduce(SemM,
                 '_/\_[T''',selectCond(selectSol(downMachineInt(T),TS))])), Atts > .

  rl [tcontinue] :
     < O : X@ToolState |
        input : 'tcont`..LotosCommand, Atts >
     => < O : X@ToolState |
           input : ('tcont_.['1.NzMachineInt]), Atts > .

  rl [tcontinue] :
     < O : X@ToolState |
        input : ('tcont_.[T]), output : nil,
        semantics : SemM,
        lotosProcess : T', transitions : TS, trace : T'', condition : T''', Atts >
     => < O : X@ToolState |
        input : 'show`state`..LotosCommand, output : nil,
        semantics : SemM,
        lotosProcess : selectProc(selectSol(downMachineInt(T),TS)),
        transitions : ttransitions(SemM,selectProc(selectSol(downMachineInt(T),TS))),
        trace : getTerm(metaReduce(SemM,
                 '__[T'',selectEvent(selectSol(downMachineInt(T),TS))])),
        condition : getTerm(metaReduce(SemM,
                 '_/\_[T''',selectCond(selectSol(downMachineInt(T),TS))])), Atts > .

  rl [show-state] :
     < O : X@ToolState |
        input : 'show`state`..LotosCommand,
        output : nil,
        semantics : SemM,
        lotosProcess : T', transitions : TS, trace : T1, condition : T2, Atts >
     => < O : X@ToolState |
        input : nilTermList,
        output :
           (meta-pretty-print-trace(SemM, T1)
            meta-pretty-print-condition(SemM, T2)
            meta-pretty-print-transitions(SemM, TS)),
        semantics : SemM,
        lotosProcess : T',
        transitions : TS, trace : T1, condition : T2, Atts > .

  rl [sat] :
     < O : X@ToolState |
        input : ('sat_.[T]), output : nil,
        lotosProcess : T', Atts >
     => < O : X@ToolState |
        input : nilTermList,
        output : if getTerm(metaReduce(['FULL], '_|=_[T',parseFormula(T)]))
                    == 'true.Bool then
                 'Satisfied '.
                 else 'Not 'Satisfied '.
                 fi ,
        lotosProcess : T', Atts > .

endm
```

5.5 The LOTOS tool environment

Input/output of specifications and of commands is accomplished by the predefined module `LOOP-MODE` [7], 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 adecuate 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.

We define in the following module 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.

```
mod LOTOS is
  pr LOOP-MODE .
  pr TOOL-STATE-HANDLING .
  pr META-LOTOS-TOOL-SIGN .
```

The state of the system is represented as a single object .

```
  subsort Object < State .

  op o : -> Oid .
  op init : -> System .

  var Atts : AttributeSet .
  var X@ToolState : ToolStateClass .
  var O : Oid .
  var Q : Qid .
  vars QIL QIL' QIL'' : QidList .
```

The `init` rule initializes the persistent object as an object of class `ToolState` by initializing its attributes.

```
  rl [init] :
     init
     => [nil,
         < o : ToolState |
            input : nilTermList,
            output : nil,
            dataExpSyntax : ['DATAEXP-SYNTAX],
            semantics : ['LOTOS-SEMANTICS],
            lotosProcess : 'stop.BehaviourExp,
            transitions : mt,
            trace : 'nil.Trace,
            condition : 'true.TransCond >,
         ('\n '\t  'LOTOS 'in 'Maude 'version '1.0
          '\n )] .
```

The `in` rule use `LOTOS-GRAMMAR` to parse Full LOTOS specifications and commands. If the input is syntactically valid, the parsed input is placed in the `input` attribute; otherwise, an error message is placed in the output channel of the loop.

```
  crl [in] :
     [QIL,
      < O : X@ToolState | input : nilTermList,
                          output : nil, Atts >,
      QIL']
     => if not(metaParse(LOTOS-GRAMMAR, QIL, anyType) :: ResultPair)
        then [nil,
              < O : X@ToolState |
                 input : nilTermList, output : nil, Atts >,
              QIL' ('ERROR: 'incorrect 'input '.)]
        else [nil,
              < O : X@ToolState |
                 input : getTerm(metaParse(LOTOS-GRAMMAR, QIL, anyType)),
                 output : nil, Atts >,
              QIL']
        fi
        if QIL =/= nil .
```

When the output attribute of the tool state contains a nonempty list of quoted identifiers, the following rule moves it to the third argument of the loop. Then the Maude system displays it in the terminal.

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

loop init .

***trace exclude LOTOS .
```

After introducing this last module the LOTOS tool is usable, and LOTOS specifications can be entered and executed, as shown in the next section.

5.6 Execution examples

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

```
Maude> (specification SPEC
type NAT is
  sorts NAT
  opns
    O : -> NAT
    succ : NAT -> NAT
    _+_ : NAT , NAT -> NAT
    eq : NAT , NAT -> Bool
  eqns
    forall N : NAT,
           M : NAT
    ofsort NAT
      O + N = N ;
      succ(N) + M = succ(N + M) ;
    ofsort Bool
```

```
      eq(0, 0) = true ;
      eq(0, succ(N)) = false ;
      eq(succ(N), 0) = false ;
      eq(succ(N), succ(M)) = eq(N,M) ;
endtype
behaviour
 h ! 0 ; stop
[]
(
   g ! (succ(0)) ; stop
 |[ g ]|
   g ? x : NAT [ eq(x,succ(0)) ] ; h ! (x + succ(0)) ; stop
)
endspec)

Maude> (show transitions .)

Trace :(nil).Trace

Condition : true

TRANSITIONS :

1. {true}{h 0}stop
2. {x = succ(0)/\ eq(x,succ(0))}{g succ(0)}stop |[g]| h ! succ(succ(0));
    stop

Maude> (cont 2 .)

Trace : g succ(0)

Condition : x = succ(0)/\ eq(x,succ(0))

TRANSITIONS :

1. {true}{h succ(succ(0))}stop |[g]| stop

Maude> (cont .)

Trace :(g succ(0))(h succ(succ(0)))

Condition : x = succ(0)/\ eq(x,succ(0))

No more transitions .
```

We have used the tool to execute larger examples, including the Alternating Bit Protocol and the Sliding Window Protocol (with more than 550 lines of code) [23]. Our tool has revealed quite efficient, giving the answer to the entered commands in few milliseconds.

We show here the specification of the Alternating Bit Protocol used.

```
(specification DataLink
   type sequenceNumber is Bool
      sorts
               SeqNum
      opns
```

```
              0      :         -> SeqNum
              inc    : SeqNum  -> SeqNum
              _equal_    : SeqNum, SeqNum -> Bool
      eqns forall x, y : SeqNum
          ofsort SeqNum
              inc(inc(x)) = x ;
          ofsort Bool
              x equal x = true ;
              x equal inc(x) = false ;
              inc(x) equal x = false ;
              inc(x) equal inc(y) = (x equal y) ;
   endtype
   type BitString is sequenceNumber
      sorts
              BitString
      opns
              empty  :                     -> BitString
              add    : SeqNum, BitString  -> BitString
   endtype
   type FrameType is Bool
      sorts
              FrameType
      opns
              info, ack  :     -> FrameType
              equal      : FrameType, FrameType -> Bool
      eqns   forall x : FrameType
            ofsort Bool
              equal(info, ack) = false ;
              equal(ack, info) = false ;
              equal(x, x) = true ;
   endtype
behaviour
   hide  tout, send, receive  in
      (  (  transmitter [ get, tout, send, receive ] (0)
        |||
            receiver [ give, send, receive ] (0)
         )
      |[ tout, send, receive ]|
         line [ tout, send, receive ] )
where
   process transmitter [ get, tout, send, receive ]
                           ( seq : SeqNum) : noexit :=
      get ? data : BitString ;
      sending [ tout, send, receive] (seq, data)
      >> transmitter [ get, tout, send, receive ] (inc(seq))
   endproc
      process sending [ tout, send, receive]
            ( seq : SeqNum, data : BitString ) : exit :=
         send ! info ! seq ! data ;
        (   receive ! ack  ! (inc(seq))  ! empty ; exit
         []  tout ; sending [ tout, send, receive ] (seq, data)
         )
      endproc
   process receiver [ give, send, receive ]
                                ( exp : SeqNum) : noexit :=
      receive ! info ? rec : SeqNum ? data1 : BitString
    ; (   [ rec equal exp ] ->
              give ! data1
            ; send ! ack ! (inc(rec)) ! empty
            ; receiver [ give, send, receive ] (inc(exp))

      []  [ (inc(rec) equal exp) ] ->
              send ! ack ! (inc(rec)) ! empty
            ; receiver [ give, send, receive ] (exp))
   endproc
   process line [ tout, send, receive ] : noexit :=

      send ? f : FrameType ? seq : SeqNum ? data2 : BitString
    ; (   receive ! f ! seq ! data2
        ; line [ tout, send, receive ]

      []  i
        ; tout
        ; line [ tout, send, receive ]
      )
   endproc
endspec)
```

6 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 [24]. 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 [24], 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 [24] 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 4.1, which we think is much more elegant and more general.

The Concurrency Workbench of the New Century (CWB-NC) [10] 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.

The Caesar/Aldebaran Development Package (CADP) [17] 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.

7 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 [24, 25], and presents several advantages as explained in Sections 2 and 6.

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 [4] 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 [5].

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. [18], 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, 2001.

[3] M. Calder and C. Shankland. A Symbolic Semantics and Bisimulation for Full LOTOS. Technical Report CSM-159, University of Stirling, 2000.

[4] M. Calder and C. Shankland. A symbolic semantics and bisimulation for Full LOTOS. In Kim et al. [18], pages 184–200.

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

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

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

[8] 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`.

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

[10] R. Cleaveland and S. T. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, Jan. 2002.

[11] 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, 2000.

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

[13] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[14] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[16] 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.

[17] 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, 1996.

[18] 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.

[19] 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.

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

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

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

[23] K. Turner. *Using Formal Description Techniques – An Introduction to Estelle, LOTOS and SDL.* John Wiley and Sons Ltd., 1992.

[24] A. Verdejo. LOTOS symbolic semantics in Maude. Technical Report 122-02, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Jan. 2002.

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

A ACT ONE and LOTOS signatures

These are the grammars used to parse ACT ONE specifications and Full LOTOS behaviour expressions.

```
fmod ACTONE-SIGN is

  sort Token NeTokenList Bubble ExpBubble .

  sorts OperDecl OperDeclList Decl DeclList TypeDecl TypeDeclList
        SortName SortNameList EqDecl EqDeclList EqDeclGroup VariDecl
        VariDeclList VariDeclGroup VarEqDeclList .

  subsort TypeDecl < TypeDeclList .
  op __ : TypeDeclList TypeDeclList -> TypeDeclList
          [assoc prec 25 gather(e E)] .

  subsort VariDecl < VariDeclList .
  op _,_ : VariDeclList VariDeclList -> VariDeclList  [assoc prec 14] .

  subsort EqDeclGroup VariDeclGroup < VarEqDeclList .
  op __ : VarEqDeclList VarEqDeclList -> VarEqDeclList [assoc prec 16] .


  subsort EqDecl < EqDeclList .
  op __ : EqDeclList EqDeclList -> EqDeclList [assoc prec 7] .

  subsort Token < SortName .
  sort TokenList .
  subsort Token < TokenList .
  op _,_ : TokenList TokenList -> TokenList [assoc prec 4] .

  subsort TokenList < SortNameList .


  subsort Decl < DeclList .
  subsort OperDecl < OperDeclList .
  op __ : OperDeclList OperDeclList -> OperDeclList [assoc prec 7] .

  op __ : DeclList DeclList -> DeclList [assoc prec 19 gather(e E)] .

  op _:_ : TokenList Token -> VariDecl [prec 10] .

  op _: ->_ : Token SortName -> OperDecl [prec 5] .
  op _:_->_ : Token SortNameList SortName -> OperDecl [prec 5] .

  op _=_; : Bubble Bubble -> EqDecl [prec 5] .
  op _=>_=_; : Bubble Bubble Bubble -> EqDecl [prec 5] .

  op forall_ : VariDeclList -> VariDeclGroup [prec 15 gather(e)] .
  op ofsort__ : Token EqDeclList -> EqDeclGroup [prec 15] .

  op sorts_ : Token -> Decl [prec 18 gather(e)] .
  op opns_ : OperDeclList -> Decl [prec 18 gather(e)] .
  op eqns_ : VarEqDeclList -> Decl [prec 18 gather(e)] .


  op type_is_endtype : Token DeclList -> TypeDecl  [prec 20 gather(e e)] .
  op type_is__endtype : Token TokenList DeclList -> TypeDecl
       [prec 20 gather(e e e)] .

endfm


fmod LOTOS-SIGN is
  pr ACTONE-SIGN .

  *** identifiers contructors
  sorts VarId SortId GateId ProcId .

  subsorts Token < VarId SortId GateId ProcId .

  sort DataExp .

  subsort ExpBubble < DataExp .

  sort BehaviourExp .

  op stop : -> BehaviourExp .
  op exit : -> BehaviourExp .
  op exit`(_`) : ExitParam -> BehaviourExp .

  sort ExitParam .
  subsort DataExp < ExitParam .  *** es asi ?
```

```
op any_ : SortId -> ExitParam .

sorts SimpleAction StrucAction Action Offer SelecPred IdDecl .

subsort GateId < SimpleAction .
subsorts SimpleAction StrucAction < Action .

op i : -> SimpleAction .
op __ : GateId Offer -> StrucAction [prec 30 gather(e e)] .

op !_ : ExpBubble -> Offer [prec 25] .
op ?_ : IdDecl -> Offer [prec 25] .
op _:_ : VarId SortId -> IdDecl [prec 20] .

op __ : Offer Offer -> Offer [assoc prec 27] .

op _`[_`] : SimpleAction SelecPred -> Action [prec 30] .
op _`[_`] : StrucAction SelecPred -> Action [prec 30] .

subsort ExpBubble < SelecPred .

*** action prefixing
op _;_ : Action BehaviourExp -> BehaviourExp [prec 35] .

*** choice
op _`[`]_ : BehaviourExp BehaviourExp -> BehaviourExp [assoc prec 40] .
            ***[assoc prec 40 gather (e E)] .
op choice_`[`]_ : IdDecl BehaviourExp -> BehaviourExp [prec 40] .
op choice_in`[_`]`[`]_ : GateId GateIdList BehaviourExp -> BehaviourExp [prec 40] .

*** parallelism
sort GateIdList .
subsort GateId < GateIdList .

subsort TokenList < GateIdList .

sort ParOp .
op |`[_`]| : GateIdList -> ParOp .
op || : -> ParOp .
op ||| : -> ParOp .
op ___ : BehaviourExp ParOp BehaviourExp -> BehaviourExp [frozen prec 40] .
op par_in`[_`]__ : GateId GateIdList ParOp BehaviourExp ->
               BehaviourExp [frozen prec 40] .

*** enable
op _>>_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .
op _>>`accept_in_ : BehaviourExp IdDecl BehaviourExp -> BehaviourExp [prec 40] .

*** disable
op _`[>_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .

*** guard
op `[_`]->_ : SelecPred BehaviourExp -> BehaviourExp [prec 40] .
```

```
  *** hide
  op hide_in_ : GateIdList BehaviourExp -> BehaviourExp [prec 40] .

  *** variable declaration
  op let_=_in_ : VarId DataExp BehaviourExp -> BehaviourExp [prec 40] .

  *** process instantiation
  sort DataExpList .
  subsort DataExp < DataExpList .

  op _`[_`]`(_`) : ProcId GateIdList DataExpList -> BehaviourExp [prec 30] .
  op _`[`]`(_`) : ProcId DataExpList -> BehaviourExp [prec 30] .
  op _`[_`] : ProcId GateIdList -> BehaviourExp [prec 30] .
  op _`[`] : ProcId -> BehaviourExp [prec 30] .


  *** process declaration

  sort ProcType .

  op noexit : -> ProcType .
  op exit : -> ProcType .

  sorts ProcDecl ProcDeclList .
  subsort ProcDecl < ProcDeclList .
  op __ : ProcDeclList ProcDeclList -> ProcDeclList
          [assoc prec 25 gather(e E)] .

  op process_`[_`]`(_`):_:=_endproc : ProcId GateIdList VariDeclList
                                    ProcType BehaviourExp -> ProcDecl [prec 20] .

  op process_`[_`]:_:=_endproc : ProcId GateIdList
                                    ProcType BehaviourExp -> ProcDecl [prec 20] .

  sort Specification .

  op specification__behaviour_where_endspec : Token TypeDeclList
       BehaviourExp ProcDeclList -> Specification [prec 50 gather(e e e e)] .
  op specification_behaviour_where_endspec : Token
       BehaviourExp ProcDeclList -> Specification [prec 50 gather(e e e)] .
  op specification__behaviour_endspec : Token TypeDeclList
       BehaviourExp -> Specification [prec 50 gather(e e e)] .
  op specification_behaviour_endspec : Token
       BehaviourExp -> Specification [prec 50 gather(e e)] .

*** FULL logic

  sort Formula .

  ops tt ff : -> Formula .
  ops _/\_  _\/_ : Formula Formula -> Formula .
  op <_>_ : SimpleAction Formula -> Formula .
  op [_]_ : SimpleAction Formula -> Formula .

endfm
```