STATE EXPLORATION BY TRANSFORMATION WITH LOLA

Juan Quemada, Santiago Pavón, Angel Fernández
Department of Telematic Engineering, Madrid University of Technology
ETSI Telecomunicacion, UPM, E-28040 MADRID SPAIN

Abstract

LOTOS is a Formal Description Technique developed within ISO to specify services and protocols. This paper describes a tool for doing LOTOS to LOTOS transformations. It has applications in state exploration, deadlock detection, testing, validation and in design by stepwise refinement. The transformations are: expansion (transformation of parallelism into summation and prefix); parameterized expansion; i action removal. The transformations obtain LOTOS specifications which relate to the original one through strong (expansion and parameterized expansion) or weak (i-action removal) bisimulation congruence.

1. INTRODUCTION

LOTOS [4] is a Formal Description Technique developed within ISO with the purpose of serving as an unambiguous language for describing service and protocol standards. Its underlying models are: a process algebra derived from CCS [6] and the abstract data type language ACT-ONE [2]. Different types of relations among LOTOS specifications are available and provide a framework for determining semantical equivalences. Observational equivalence [6], testing equivalence [7] and implementation relation [1] seem to be the most interesting ones. Equational properties, expansion theorems and other mathematical transformations existing for the equivalences and relations mentioned give a sound mathematical basis for performing LOTOS to LOTOS transformations which preserve equivalence or relationship between input and output.

2. DESCRIPTION OF LOLA

The LOLA (LOtos LAboratory) system is a transformational tool developed at the Department of "Ingeniería Telemática" of the Technical University of Madrid for serving as an experimental environment. An earlier version is described in [8]. It has been designed in Pascal. The current version runs on SUN systems. At the beginning the tool was conceived for verification on labeled transition systems generated from a LOTOS specifications, but due to the state explosion problems appearing, compacting and parametrization was introduced as a way to make work with bigger specifications possible. Three transformations have been implemented in LOLA by now and others are under study. They are: expansion, parameterized expansion, and i-action removal.

LOTOS is based on the mixture of two different models, algebraic calculi of processes with an operational semantics and equational abstract data types with an algebraic initial semantics. Thus the tool gives a different treatment to both parts. The data values are treated operationally by interpreting equations as rewrite rules from left to right. Thus the tool does not process LOTOS, but an operational version of it where equations interpreted from right to left shall be a set of rewrite rules. These rewrite rules should be confluent and terminating for achieving proper operation. Some Knuth-Bendix completion algorithm, like for

example [3], is necessary for making the data type definition operational. Data values or expressions containing variables are represented always through their normal forms.

Semantic equality exist in LOTOS as an user operation and may be used freely in specifications. Semantic equality (=) is treated as a predefined boolean function, called `equal`. Syntactic equality of ground terms after rewriting right and left sides is used. Extra rewrite rules can also be given to the it. This treatment is complete only for ground terms. It implies working only in the equational theory of the given data types.

LOLA deals the expressions symbolicly in many situations where some variables remain still undefined. Symbolic processing of such expressions is equivalent to theorem demonstration. LOLA needs to know if the predicates in guards, selection predicates or arising from synchronization conditions are equal to false or true for any possible value of its variables, or if if depends on the variables. The theorems of the equational theory of a data type will be found by LOLA, but not the ones of the inductive theory of a data type. Rewriting is not enough for demonstration. Induction may be needed in some cases and undecidability may exist in others Semiautomatic systems may be used for inductive theorems demonstration, like [3] through inductionless-induction or others. In the following, except where strictly necessary, all the problems caused by the incompleteness of rewriting with respect to the theorems of the inductive theory will not mentioned.

The impact of this on the transformed output of the tool is the following. Transformations produced by the tool will de correct, but will be not optimized in the following sense. Some guards and some selection predicates which are equivalent to true or false have not been removed (true case) or substituted by a stop (false case) because rewriting has not been enough for determining it.

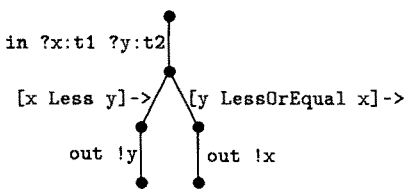
3. EXPANSIONS: STATE EXPLORATION

The semantics of LOTOS is defined operationally as a labeled transition system, making state exploration straightforward. It is just the calculation of the labeled transition system generated by a specification. The plain state space is usually enormous even for very simple specifications. This paper describes some ways of reducing the state space by using some form of symbolic representation of the transitions. Two reduced explorations are presented, which have been called the expansion and the parameterized expansion. The first one keeps the variable definition ("`event ?x:t`" .., and "`choice x:t..`") of a specification as such without expanding to the choice of all the possible values of the sort. The second reduces the state exploration to the basic behaviour states, parameterized by generic instances of the variables of the specification, keeping also the variables in a symbolic way.

Equational properties and the expansion theorems existing in LOTOS provide the basis for performing the state exploration as a LOTOS to LOTOS transformation. This is very useful from the engineering point of view because it allows an easy reuse of the resulting exploration because its just LOTOS. The transitions of a labeled transition system can be represented in LOTOS as events which offer only data values (this means no variables). Action prefix ";" and choice "[]" operators are also needed to represent the ordering in time of the

transitions. With this components trees can be constructed. Loops can be created by process definition and instantiation.

As the representation of variables is sometimes symbolic in both expansions, the restrictions to be hold by this variables must be kept in the transition systems. LOTOS guard statement "`[ boolean-expression ] ->`" and selection predicates "`event ?x:t .. [ Predicate(x) ]`" allow the representation of such restrictions. The example below shows graphicly such type of conditions, where predicates are represented as labels of transitions also. This special transitions are not real transitions, just conditions on the following ones. This examples shows also how the variables are kept in the compressed transition system. We will call this transitions, extended transitions. Such trees must have a unique name for every variable across all the tree.

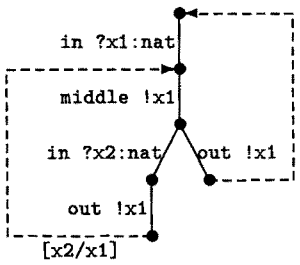



```
  in ?x:t1 ?y:t2
; (   [ x Less y ] ->
        out !y
      ; stop

  []  [ y LessOrEqual x ] ->
        out !x
      ; stop
)
```

Figure 2: Compressed Transitions

There exist LOTOS specifications which will have an infinite transition system also in the compressed representation, but in many cases of interest a finite representation can be found by detecting duplicated behaviours. LOTOS specifications which have a finite number of extended transitions going out of any state of his transition system expressed in such a compressed way and a finite number of states, have a finite representation in terms of such constructs by duplicate behaviour detection. Due to the finiteness of a specification and due to its structure, we will always find after a finite number of transitions, a "`stop`" like state, an "`exit`" transition or a behaviour which is exactly equal to one existing in the same path except for some variable names, or for some values associated with variables of the behaviour. In the three cases we can stop the exploration and abstract all the future behaviour by some LOTOS construct, "`stop`", "`exit`" or process definition and instantiation. In the third example the specification buffer2 has two infinite processes (one element buffer cells) are composed in parallel to form a two element buffer. It shows how the duplicate behaviour detection is done. In the figure the dashed lines relate the duplicated states in the tree (loops). This corresponds to process instantiation and definition in LOTOS. The right dashed line corresponds to DupBeh1 and the left one to DupBeh2. DupBeh2 has variable renaming but no parametrization.

```
SPECIFICATION buffer2 [in, out, middle] :noexit
BEHAVIOUR
    (buffer1 [in ,middle] |[middle]| buffer1 [middle, out])
WHERE
   PROCESS buffer1 [in, out] :noexit :=
      in ?x:data ; out !x ; buffer1 [in, out]
   ENDPROC

TYPE data IS .... ENDTYPE
ENDSPEC
```




```
Input Specification
PROCESS DupBeh1 [in, out, middle] :noexit :=
   in ?x1:data
  ; DupBeh2 [in, out, middle] (x1)
ENDPROC

PROCESS DupBeh2 [in ,out, middle]
(x1:data):noexit:=
   middle !x1
  ; (   out !x1 ;
      ; DupBeh1 [in, out, middle]

    []  in ?x2:data
      ; out !x1
      ; DupBeh2 [in, out, middle] (x2)
    )
ENDPROC
```

Figure 3: Output Specification

3.1. EXPANSION

Expanding a specification means removing operators, such as parallelism "| [gates] |" enabling ">>" or disabling "[>", substituting them for the equivalent transition system. This is done by the application of the so called expansion theorems [4]. This transformation generates a new specification in a subset of LOTOS which includes only the following operators : actions denotations, action prefix ";", alternative "[]", choice statements "choice x:t []", guards "[Predicate] ->", process definition and instantiation. The transformed specification is strong bisimulation equivalent with the original. There exist unresolvable guards which may depend on values supplied by the environment. This implies that not all the guards can be removed during the expansion.

The generation of the expansion stops when a "stop" is found, an "exit" statement is found or a behaviour is found which is exactly equal (syntacticly) to a previous one, for each branch of the expansion. When a duplicated behaviour is found the first one is transformed into a process instantiation-definition pair, and the duplicated one into a process instantiation one. For the termination of the expansion only finite sorts of data values and bounded parallel, enabling or disabling compositions are allowed. As a general rule, any behaviour expression that produces infinite transitions starting from a state or with an infinite number of states can not be expanded. As a simple example of such a transformation lets see the following specification.

```
SPECIFICATION Loops [ one, two] :noexit
BEHAVIOUR     (loop[one](0) ||| loop[two](0))
WHERE
   PROCESS loop[a](n:nat2):noexit := a!n;loop[a](succ(n)) ENDPROC

   TYPE Nat2 IS
      SORTS   nat2
      OPNS    0     :       -> nat2
              succ : nat2 -> nat2
      EQNS    FORALL x:nat2 OFSORT nat2      succ(succ(x )) = x ;
    ENDTYPE
 ENDSPEC
```

The application of the expansion to this specification produces the result presented below. Notice that the natural number is modulo two such that the expansion finishes quickly. In case of having just natural number the expansion of this specification would not have stopped.

```
specification loops [one, two] :noexit
behaviour    ProDup0 [one, two]
where
process ProDup0 [one, two] :noexit :=
   (one!0;ProDup2[one,two] [] two!0;ProDup3[one,two])
endproc
process ProDup1 [one,two] :noexit :=
   (one!succ(0);ProDup3[one,two] [] two!succ(0);ProDup2[one,two])
endproc
process ProDup2 [one,two] :noexit :=
   (one!succ(0);ProDup0[one,two] [] two!0;ProDup1[one,two])
endproc
process ProDup3 [one,two] :noexit :=
   (one!0;ProDup1[one,two] [] two!succ(0);ProDup0[one,two])
endproc
endspec
```

The expansion can have applications in deadlock detection, in testing, in putting up a testbed for abstract specifications and used also for simulation of behaviours. Testing consists, in LOTOS, in specifying a test process or a test sequence (also in LOTOS) and obliging it synchronize with the specification under test. The result of the expansion is the results of the test. A test is usually a sequence or a tree of LOTOS events terminating with a success indicating event, which of course must be different from any existing event of the specification under test. An example of such a test composition is shown below.

```
(              SpecificationUnderTest [<events>]
|[<events>]|   Test [<events>, SuccessEvent]
)
```

3.2. PARAMETERIZED EXPANSION

The parameterized expansion is a variation of the previous one, with two differences. The treatment of the expansion finalization and the treatment of the value expressions which is done always symbolicly. Now the expansion stops when a "stop", an "exit" or a parameterized behaviour of an existing one is found. By parameterized behaviour of an existing one we mean a behaviour which is exactly equal to the previous one except for some value expressions. The state exploration done is thus limited. Only states parameterized by generic instances of the variables are visited.

The only ways for the data values to affect the expansion is through guards, synchronization and selection predicates, thus the parameterized expansion must keep all the possible actions of every parameterization. They shall be treated according to this and must be rewritten to a generic normal form but without

doing any substitution of variables for values such that they are still valid for every possible instantiation. The values are used only for generating the actual parameters of the process instantiations.

The divergence condition for the parameterized expansion is different. Infinite sorts are allowed now but with respect to dynamic creation of processes the conditions are more restrictive. As a general rule, any behaviour expression that produces infinite transitions (from the syntactical point of view) from a particular state or has an unbounded number of parameterized states will diverge.

When the parameterized expansion is applied to the previous example, we obtain only the parameterized state exploration. As soon as a behaviour is detected which is equal to a previous one except for parameter instances, the expansion stops. This expansion would have produced the same result in case of having infinitely many different values of sort "nat2".

```
specification loops [one, two] :noexit
behaviour     ProDup0 [one, two] (0, 0)
where
process ProDup0 [one, two] (nv_36:nat2, nv_37:nat2) :noexit :=
   (  one !nv_36 ; ProDup0 [one, two] (succ(nv_36 ),nv_37)
   [] two !nv_37 ; ProDup0 [one, two] (nv_36,succ(nv_37))
   )
endproc
endspec
```

One important use of the parameterized expansion is the derivation of efficient implementations. Usually protocol implementations in operating systems, kernels and/or imperative languages are state machines which can be quiet large but efficient in actual Von Neumann type processors. The result of parameterized expansion is just a state machine equivalent to extended automatons used in state descriptions and implementation of protocols. In fact the expansion process is just a precalculation of all the synchronizations possible for some specification, removing the overhead of having to calculate it during runtime.

3.3. DEADLOCK DETECTION ON EXPANDED SPECIFICATIONS

Deadlock detection can be done in any of the expanded forms. In fact they have been calculated during the expansion. There will be explicit deadlocks (actions leading to lonely stops, which means that there are no outgoing transitions from those states) and invisible potential ones (in all the guards and selection predicates which may make a state equivalent to a lonely stop for some data values). The potentially invisible deadlock detection can suffer from the limitations of rewriting, because guards and selection predicates can be explicit deadlocks not detected by rewriting the value expressions. Other properties as unproductive loops, recoverability, ... can be studied on the expanded form.

4. i-LOOP REMOVAL

This transformation has been conceived to be applied to the output of the expansions, with the purpose of reducing redundant i actions generated during the expansions and thus reducing its size. The result of the expansions has a tree like form. The initial state is the root and some process instantiations can

be considered as leaves with recursion (or jumps) to parts of the fundamental behaviour which is the tree.

We call "i-loops" to process definitions which have a path in the tree which starts at a definition and ends which a call to the same definition formed only of "i" actions, action prefix, alternative and process definitions or instantiations. Such loops represent divergent behaviour and may be potentially dangerous or recovering depending on the interpretation. Weak bisimulation can not differentiate both cases. But they can be removed according to it. The transformation consists in the elimination of all the internal actions of the loop. This transformation applies also other reductions like: a; i; B $\Rightarrow$ a; B and stop [] B $\Rightarrow$ B. The internal action loop removal can not be represented by equations. The process done is graphically represented in figure 4.

Original behaviour Transformed behaviour

Figure 4: "i-loop removal"

5. CONCLUSIONS AND FURTHER WORK

The expansions do state exploration and are thus quite computing intensive. One practical limitation is related to the performance that can be achieved in terms of speed and memory. The actual version of the tool is a research prototype, but in spite of this, fairly large specifications have been run on it. Improvements in term of performance seems possible. Actual speed is around 50 states per second and may degrade with the type of specs. On the other hand, the reduction of the state space (to be explored) obtained by parameterizing it, allows the expansions of more complex systems. This brings this tool close to real systems design. More experience and more efficient version of the tool are needed to know how far we can get.

Another limitation arises from the fact that only specifications which have a transition system with a finite representation in the output representation form can be expanded. As finite systems (except for very pathological cases) can be expanded this limitation seems not to be severe. On the other hand the parameterized expansion can deal with infinite data types extending its applicability to some types of infinite systems.

The following developments are considered of interest: 1) Identification of new transformations, like (pseudo) canonical or minimal forms or ways of interchanging operators. 2) The introduction of time following [5]. 3) Extension of a modal reasoning to work with data values and parametrization.

APPENDIX A. THE LANGUAGE OF LOLA

The internal language in which LOLA works is a simplified version of LOTOS in which the complications (overloading, nesting,..) of the static semantics are removed by using unique naming. At this level a specification will be a

BehaviourExpression a list of *ProcessDefinitions* and a plain *TypeDefinition*. The language is described in the table below. Angle brackets means optional. The the LOTOS statements not existing are translated into equivalent in it. The semantics is similar to equivalent LOTOS constructs.

Name	Syntax of Behaviour Expressions
inaction	*stop*
action prefix	$g < d_1..d_n >< [BE] >; B$
	$i; B$
termination	$exit < (E_1, ..., E_n) >$
choice	$B_1 [] B_2$
parallel	$B_1 \|[g_1, ..., g_n]\| B_2$
hiding	hide $g_1, ..., g_n$ in B
process definition	P $[g_1, ..., g_n] < (x_1 : s_1, .., x_n : s_n) >:= B$
process instantiation	P $[g_1, ..., g_n] < (E_1, ..., E_n) >$
relabelling	$B[g_1/g'_1, .., g_n/g'_n]$
enabling	$B_1 >>$ <accept $x_1 : s_1, ..., x_n : s_n$ in> B_2
disabling	$B_1 [> B_2$
guard	$[BE] - > B$
sum-expression	choice $v_1, .., v_n [] B$

APPENDIX B. THE DEFINITION OF THE PARAMETERIZED EXPANSION.

The parameterized expansion is a generalization of the expansion theorems, by allowing symbolic data values. Only the parallel composition is shown, although expansions for enabling and disabling exist with a similar generalization in which the variable definition, guards and selection predicates are maintained.

The expansion is defined for behaviours containing only sumexpresions (*choice* $x_1 : t_1, \ldots, x_n : t_n$ [] B), choice expression (*B1* [] *B2*), action prefix (*a; B*), guarded expression ([e]-> B) and action denotations with value offering ($g\ !v_1\ !v_2\ \ldots !v_n$). This formulation is general because other LOTOS constructs can be expressed in term of these ones like, value acceptance, selection predicates, ... ($g?x : t[P]; B$ is the same as choice $x : t[]([P] - > a!x; B)$). LOLA uses a more efficient version of it in which specific rules exist for every LOTOS construct, which does not introduce any new idea but is much longer.

Some preprocessing, done by rewriting, is needed to prepare behaviours for the parameterized expansion. Standard equations of LOTOS defined in (Annex B) of [4] are used: *choice 3, hiding 5a, 5b, 6, 8, 9, 10, guarding 1a, 1b, instantiation 1, local definition 1, relabelling 1, 2, 3, 4, 6, 7, 8, 11.* Other rewrite rules used, which are not in the standard are:

choice v [] (choice v' [] B) $\Rightarrow$ choice $v.v'$ [] B,
[e]->([e']->B) $\Rightarrow$ [e.e']->B,
[e]->(choice v [] B) $\Rightarrow$ choice v [] ([e]->B),
choice v [] ($B1$ [] $B2$) $\Rightarrow$ (choice v [] $B1$) [] (choice v [] $B2$),
[e]->($B1$ [] $B2$) $\Rightarrow$ ([e]->$B1$) [] ([e]->$B2$)

Where the operation "·" stands for joining boolean expressions or choice expressions (for example $(x = y) \cdot (suc(x) + y = x) \equiv ((x = y) and (suc(x) + y = x))$

(remember that $=$ is here a boolean function) and ($choice\, x : t_1 [] \cdot choice\, y : t_2 \equiv choice\, x : t_1, y : t2 [])$) and the symbol $\Rightarrow$ stands for transform into.
The generalized expansion theorem is as follows: Let $B_1 = \sum_i CH_i\ G_i\ a_i; B_i$, $B_2 = \sum_j CH_j\ G_j\ a_j; B_j$, and $a_i = g\ !v_1 \ldots !v_n$, $a_j = g'\ !v'_1 \ldots !v'_n$; then

$$
\begin{array}{lll}
B_1 \mid [A] \mid B_2 = & & \\
\quad \sum_i\ CH_i\ G_i\ a_i; (B_i \mid [A] \mid B_2) & \forall\, i \mid \text{gate}(a_i) \notin A \\
[] \quad \sum_j\ CH_j\ G_j\ a_j; (B_1 \mid [A] \mid B_j) & \forall\, j \mid \text{gate}(a_j) \notin A \\
[] \quad \sum_{i,j}\ CH_{i,j}\ G_{i,j}\ a_j; (B_i \mid [A] \mid B'_j) & \forall\, i,j \mid \text{gate}(a_i) = \text{gate}(a_j) \\
 & \wedge\ \text{gate}(a_i) \in A
\end{array}
$$

where $CH_{i,j} = CH_i.CH_j$, $G_{i,j} = G_i.G_j.E$, $E = (v_1 = v'_1, \ldots, v_n = v'_n)$ and $B'_j = B_j[v_1/v'_1, \ldots, v_n/v'_n]$ being CH_i, CH_j choice expressions, G_i, G_j guarded expressions and B_i, B_j behaviours expression. Its demonstration is straightforward. Exactly the same transitions are generated by both parts.

References

[1] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS Specifications, Their Implementation and Their Tests. In *Sixth International Workshop on Protocol Specification, Testing and Verification*, Montreal, June 1986.

[2] H. Ehrig, W. Fey, and H. Hansen. *ACT ONE: An Algebraic Language with two Levels of Semantics.* Technical Report, Tech. Universitat Berlin, 1983.

[3] R. Foorgard. *Reve-A Program for Generating and Analyzing Term Rewriting Systems.* Technical Report MIT/LCS/TR-343, September 1984.

[4] ISO. *LOTOS a Formal Description Technique based on the Temporal Ordering of Observational Behaviour.* IS 8807, TC97/SC21, 1989.

[5] D. Frutos J. Quemada, A. Azcorra. *A Timed Calculus for LOTOS.* Technical Report, March 1989.

[6] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, Berlin, 1980.

[7] R. Nicola and Hennessy, M.C.B. Testing Equivalences for Processes. *Theoretical Computer Science*, 34(1,2):83–133, Nov 1984.

[8] J. Quemada, A. Fernandez, and J.A. Manas. LOLA: Design and Verification of Protocols using LOTOS. Ibericom, Conf. on Data Communications, Lisbon, May 1987.