References

[BB] Bolognesi, B. and Brinksma, E. Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14 (1987) 25-59.

[BJ] Brand, D., and Joyner, W.H. Jr. Verification of Protocols Using Symbolic Execution. Computer Networks 2, 4/5, 351-360.

[BSS] Brinksma, E., Scollo, G., and Steenbergen, C. LOTOS Specifications, their Implementations, and their Tests. In: B. Sarikaya and G.v. Bochmann (eds.) Protocol Specification, Testing, and Verification, IV. North-Holland, 1987, 349-360.

[DEM] de Meer, J. Derivation and Validation of Test Scenarios Based on the Formal Specification Language LOTOS. In: B. Sarikaya and G.V. Bochmann (eds.) Protocol Specification, Testing, and Verification, VI. North-Holland, 1987, 203-216.

[EB] Eertink, E., and Brinksma, E. Implementation of a Test Derivation Algorithm. Technische Hogeschool Twente, Oct. 1987 (SEDOS/C2/N82).

[FL] Favreau, J.P., and Linn, J.R. Automatic Generation of Test Scenario Skeletons from Protocol Specifications written in Estelle. In: B. Sarikaya and G.V. Bochmann (eds.) Protocol Specification, Testing, and Verification, VI. North-Holland, 1987, 191-202.

[GHL] Guillemot, R., Haj-Hussein, M., and Logrippo, L. Executing Large LOTOS Specifications. University of Ottawa, Department of Computer Science, Technical Report 88-03 (Jan. 1988).

[ISO1] International Organisation for Standardization. Information Processing Systems. Open Systems Interconnection. LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior (ISO DIS 8807), 1987.

[ISO2] International Organization for Standardization. Formal Description of ISO 8072 in LOTOS. (ISO/TC 97/SC 6/WG 4/N 317), 1987.

[LP] Logrippo, L. and Probert, R.L. Protocol Specification-Level Validation. In: Sunshine, C. (ed.) Protocol Specification, Testing, and Verification North-Holland, 1982, 303-304.

[MY] Myers, G.J. The Art of Software Testing. Wiley, 1979.

[SBMS] Sarikaya, B., Bochmann, G.v., Maksud, M., and Serre, J.M. Formal Specification Based Conformance Testing. In: Communications Architectures and Protocols, SIGCOMM ’86 Symposium, 236-240.

[UR] Ural, H. A Test Derivation Method for Protocol Conformance Testing. In: H. Rudin and C.H. West (eds.) Protocol Specification, Testing, and Verification, VII. North-Holland, 1987, 347-358.

[URS] Ural, H., and Short, R. An Interactive Test Sequence Generator. In: Communications Architectures and Protocols, SIGCOMM ’86 Symposium, 241-250.

[VE] Van Eijk, P. Software Tools for the Specification Language LOTOS. University of Twente, 1988.

obtaining real test specifications with values, mentioned in Section 4.

Finally, while the main emphasis of this paper is on testing, it sould be noted that trees are also interesting in verification. Therefore, application of similar techniques in verification appears to be possible.

Acknowledgment. The interpreter was written by J.P. Briand, M.C. Fehri, R. Guillemot, and M. Haj-Hussein. We are indebted to A. Obaid for many useful discussions, and we have also used some ideas due to H. Elgendy. This work was supported in part by the National Science and Engineering Research Council of Canada and Bell-Northern Research.

- and then constructing the expanded tree.

Concerning the first step, this consists in replacing every symbolic action, containing or not a guard, by values. The set of values that can be accepted by an action is usually infinite.

Example:

For the LOTOS specification :

```
g?x:Nat ; g?y:Nat [y gt x] ; exit
```

the SST obtained is:

```
1  g?Nat@1
|  1  g?Nat@2  [Nat@2  gt  Nat@1 ]
|  |  1  exit
```

Values for *Nat@1* and *Nat@2* must be chosen before this sequence can be used as a test case. Possible values belong to the set of all pairs *(x,y)* of natural numbers such that $x<y$. Strategies for choosing such test values have been studied in the testing literature and will not be discussed in this paper [MY].

5. Conclusions and Future Work

The work presented in this paper is an effort towards a methodology for generating test suites from LOTOS specifications. We showed that useful execution trees can be generated by using existing interpreters and some basic simplification rules. More sophisticated heuristics could be added to the system. Similarly, other congruence rules could be added to the very basic list given in Section 3.2. As a further step, we are envisaging the use of theorem-proving methods to enhance the methods for detecting contradictions and equivalences.

Furthermore, this method should be related to the existing theory on generating test suites from LOTOS specification [EB][BSS]. And then, there is the problem of

Internal events could be eliminated at the source by using modified inference rules, such as (in simplified form):

exit -exit-> stop

A >> B -a-> A' >> B if A-a-> A' and name(a) =/= exit

A >> B -a-> B' if A -exit-> A' and B-a-> B'

Unfortunately, these rules do not work in some cases. For example, suppose that the *exit* statement appears as the first action as in the following example:

```
process p[a,b] :=  exit >> a ; stop
                   []
                   b ; stop
endproc
```

this behavior is equivalent to: *i ; a ; stop [] b ; stop.* By eliminating the internal event according to the rules above, we obtain the behavior:

a ; stop [] b ; stop.

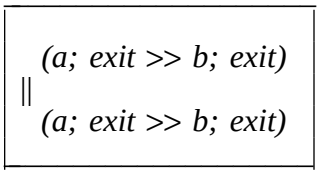
This expression is not equivalent to the previous one. The semantics of the original specification give priority to *b* while the semantics of the second specification don't. The modified rules for the exit can only be used if constructs such as the one in the example do not occur in the specification, and this can be checked statically.

The question of the treatment of internal events due to enabling for testing purposes deserves further study.

4. Considering Values

After obtaining the simplified tree according to the procedure described above, execution sequences can be derived by the following steps:

- identifying, for every action in all remaining paths, all the values that can be accepted



If we call *i1* and *i2* the internal actions resulting from the first and second enable respectively, the resulting **LST** shows both mutual orderings of these actions, i.e.

```
1 a
| 1 i1
| | 1 i2
| | | 1 b
| | | | 1 exit
| 2 i2
| | 1 i1
| | | 1 b
| | | | 1 exit
```

The tree of observable actions instead is simply

```
1 a
| 1 b
| | exit
```

By applying the simplification rules discussed in **3.2.a)** we get:

```
1 a
| 1 i
| | 1 b
| | | exit
| 2 i
| | 1 b
| | | exit
```

This tree cannot be simplified further by the look-ahead mechanism because the behavior expression resulting from the execution of *i1* is

(b; exit) || (exit >> b; exit)

while the behavior expression resulting from the execution of *i2* is

(exit >> b ; exit) || (b; exit)

and unfortunately these two behavior expressions are not textually identical.

The internal events will be completely eliminated by using the algorithm in 3.2.b) (and, of course, could be eliminated by the look-ahead mechanism if we informed it that $p||q = q||p$).

```
a ; b ; c ; d
```

by applying first rule 2, and then rule 1. Annex 2.2. shows an extended example.

b) **Application of Congruence Rules on the Resulting Tree**

The tree resulting from the simplification process described above is stored in memory and further simplified by an algorithm that is able to detect other cases in which rules 1 to 3 can be applied. Consider for example the following behavior expression:

```
a ; b ; (c ; d ; stop
        []
         i ; c ; d ; stop)
[]
 a ; b ; c ; i ; d ; stop
```

By using the "look-ahead" mechanism, the following tree will be saved:

```
   a ; b ; c ; d ; stop
[]
   a ; b ; c ; d ; stop
```

By scanning the tree according to this algorithm, it is simplified to:

```
a ; b ; c ; d ; stop
```

3.3. The Enable Operator.

In our study of realistic examples, it soon became obvious that some method had to be found to manage the complexity generated by internal events due to enables. This can be seen by a study of Annex 3.2. For example, consider the following behavior expression:

Note that the second rule is a stronger version of the well-known congruence *B[]i;B = i;B.* It is more useful than the latter, especially when disable is present.

The most obvious way to perform reduction by congruence rules is to generate the whole tree, to store it in memory, and then to scan it bottom-up to find places where congruence rules can be applied. However, memory can be saved if congruence rules are applied as far as possible, by using a look-ahead mechanism, already while the tree is being generated. The stored tree is then scanned bottom-up to further simplify it.

a) **Application of Congruence Rules While Building the Tree**

The interpreter is unable to directly apply rules 1 to 3 above. All it can do is to to compute sets of possible next actions with resulting behavior expressions. Situations where these simplifications can be applied are detected by a "look-ahead" mechanism. Consider for example rule 2. When the interpreter finds that the set of possible "next actions" is of the form N = {a1,...,am,b1,...,bn,i}, if the set of next behaviors is respectively {A1,...,Am,B1,...,Bn,D}, the set of next actions for D is computed. If this set includes {b1,...,bn}, with next behaviors {B1,...,Bn} respectively, then the tree is simplified by using only the actions in {a1,...,am,i} and their successors. Again, two behavior expressions are considered to be the same only if they are the same character by character.

A corresponding criterion is used in testing for equivalence for rule 3.

These rules are applied recursively while possible. Therefore, an expression such as

```
a ; b ; (c ; d ; stop
         []
          i ; c ; d ; stop)
```

is reduced to

3.1. Internal Events and Implementations

Internal actions introduce nondeterminism. Implementations may differ by the way they reduce this nondeterminism. Thus, for a given specification, one can obtain several valid implementations [BSS]. For instance, consider the following process:

```
process Connection[ConReq,DisInd,ConConf] : exit :=
   ConReq ; (  ConConf ; exit
            []
               i ; DisInd ; exit )
endproc
```

This is the connection phase of a protocol that always accepts a disconnection indication after a connection request, but may refuse the connection confirmation. The choice between these two alternatives is left to the implementation. Therefore, there are three possible implementations for this specification. One is the specification itself. The other two are:

- *ConReq ; (ConConf ; exit [] DisInd ; exit)*

- *ConReq ; (DisInd ; exit)*

The first alternative always offers *ConConf*, while the second nevers offers it.

Internal events designating implementation choices cannot be eliminated from the tree.

3.2. Simplification by Congruence Rules

In some cases, internal events can be removed by applying congruence rules. This removal does not in any way change the semantics of the specification. In this experiment, we implemented only the following rules:

1	*a;i;B*	is simplified to	*a;B*
2	*B [] i;(B[]C)*	is simplified to	*i;(B[]C)*
3	*B[]B*	is simplified to	*B*

variable *x* would not be evaluated and would therefore all be considered to be true, unless they contain some contradiction independent of the value of *x*.

While equivalence of behavior expressions is an undecidable problem, more sophisticated criteria of behavior equivalence could be added to our system, also in consideration of the needs of the application. For example obviously behavior *a [] b* can be considered to be identical to behavior *b [] a*. Furthermore, it is well-known that for testing purposes it may be appropriate to consider equivalent behaviors that cannot be considered to be equivalent from other points of view.

b) **Ignoring Some Paths**

In generating behavior trees for complex systems, it is normal that the user may wish to ignore certain paths. For example, this can happen for paths relating to error conditions, or for paths relating to the creation of several connections if it is wished to consider the case of one connection only. Such paths are usually guarded by internal actions. Our system allows one to specify that the entire subtree following a certain internal action be ignored.

3. The Treatment of Internal Events

A process in LOTOS is described in terms of its actions, which can be of two types: observable actions or internal actions. Internal actions occur in execution sequences either because they are specified explicitly (an *i* in the specification) or because they result from the dynamic behavior of the system (we call this implicit specification). This is the case for example when the enable (>>) operator is used together with the *exit* statement.

Internal events, especially those due to enable operations, are a major cause of complexity in the symbolic tree. Hence the importance of eliminating them when possible.

2.3 Towards a Limited Tree

Trees generated by this method are usually infinite. This is the normal case when recursion is involved. Two methods of dealing with infinite paths are detecting recursion, and ignoring some paths under user control.

a) **Detecting Recursion**

Recursion can be detected automatically at least in some cases. For example, a unique identifier can be associated with an occurrence of a behavior expression in a tree. Later occurences of the same behavior expression or of an equivalent one in the same path are then replaced by the identifier preceded by the word "again". This can be done to a certain extent while building the tree, by comparing each behavior obtained against the ones obtained previously. The currently used comparison criterion is strict character-by-character identity. Although this may appear to be an overly simple criterion, we have found that it is useful in many cases. This is shown in Annex 2.2.

Example:

```
process P[a,d]:exit :=   a ; P[a,d]
                       []
                         d ; exit
```

The LST is:

```
1 a
| 1 a
    ...
| 2 d
| | 1 exit     EXIT
2 d
| 1 exit     EXIT
```

while the SST is:

```
bh0   1 a      ==> again bh0
      2 d
      | 1 exit EXIT
```

Also, according to our criterion a behavior expression such as *P[a](x)* is considered identical to *P[a](succ(x))*, while *P[a](0)* would be considered different from *P[a](succ(0))*. This is because in the second case some predicates will be yielding different values for *0* and *succ(0)*, while in the first case, predicates involving the

In addition, our system allows the user to establish a data base of contradictions. A user-defined contradiction can involve several terms.

Example:

```
( in?x:Nat [x gt 3] ; out?y:Nat [y gt x] ; exit
[]
  in?x:Nat [x le 3] ; out!3 ; exit  )
||
( in?x:Nat; (out?y [y lt 3] ; exit
            []
            out?y [y eq x] ; exit    )  )
```

Assume that the data base of contradictions contains:

(1) [x gt y] # [x eq y]
(2) [x gt y] & [y gt z] # [x lt z]

Before simplification, the tree is:

```
1 in?Nat@1 [Nat@1 gt 3]
| 1 out?Nat@2 [Nat@2 gt Nat@1] [Nat@2 lt 3]
| | 1 exit     EXIT
| 2 out?Nat@2 [Nat@2 gt Nat@1] [Nat@2 eq Nat@1]
| | 1 exit     EXIT
2 in?1Nat@1 [Nat@1 le 3]
| 1 out!3 [3 lt 3]
| | 1 exit     EXIT
| 2 out!3 [3 eq Nat@1]
| | 1 exit     EXIT
```

- Branch 1.1 is pruned because of D(2) (it implies 3 < 3).

- Branch 1.2 is pruned because of B(1).

- Branch 2.1 is pruned because of A (the predicate contains no variables and can be evaluated to false).

The SST is:

```
1 in?Nat@1 [Nat@1 gt 3]    DEADLOCK
2 in?Nat@1 [Nat@1 le 3]
| 1 out!3 [3 eq Nat@1]
| | 1 exit     EXIT
```

value *y* bound at gate *g* must be less than value *x* chosen arbitrarily by the environment. Since a *choice* is not an action, we use the symbol % and represent *x* by *Nat%1*.

The LST is: 1 g?y:Nat [y lt x]

The SST is: 1 g?Nat@1 [Nat@1 lt Nat%1]

2.2. Feasible Symbolic Trees

One may eliminate certain paths that are not feasible, by trying to evaluate symbolically guards and selection predicates [BJ]. Predicates that cannot be evaluated to false and are not in contradiction with others previously assumed to be true are assumed to be true. Each action is associated with a list of predicates, which gives all the constraints that must be satisfied for the action to be executed (these are the combined selection predicates of all action offers cooperating in the action). We call these *action predicates.* It is also associated with the list of predicates that occurred ahead of it on the same path, in guards or other predicates. We call these *path predicates.*

During the tree building process, an action is reduced to a *stop* if a contradiction is detected in its path predicates. It is checked in the following order whether:

A) One of the action predicates can be evaluated to false.

B) A contradiction can be detected in the action predicates

C) A contradiction can be detected in the path predicates.

D) A contradiction can be detected between path predicates and action predicates.

The detection of contradictions in the general case is of course an undecidable problem. Some heuristics are needed. Contradictions such as *(q(x) and p(x))*, where *q(x) = not(p(x))* appears in the list of axioms, are detected automatically. Upon finding a predicate such as this one, the system scans the list of axioms looking for such immediate contradictions. In specifications we have studied [ISO2], such cases are frequent.

2. Obtaining a Significant Symbolic Tree

2.1 Contextual Symbolic Trees

Actions specified for a process may contain variables to be bound by the environment, values to be offered to the environment, and conditions on the variables and values (guards and selection predicates).

We use a symbolic representation for the variables which allows us to relate several occurences of the same variable with different external names. A renaming scheme is used. Each occurrence of a variable is replaced by an identifier (a "symbol") which consists of:

- the variable's sort,
- a symbol which expresses how the value was bound, i.e. @ for a variable bound at a gate, and % for a variable bound in a *choice,* an *exit(any),* or an initial process parameter.
- an identifier that shows the depth in the tree of the variable's first occurrence.
- a second identifier to distinguish different variables of the same sort and the same nesting level (if needed).

Example:

```
g?x:Nat ; exit(x)   >>   accept y:Nat in   g!y ; stop
```

The value of variable *x* is exported (by means of *exit* and enable operators) to become bound to variable *y.* By the renaming process, both variables get the indentifier *Nat@1,* which stands for variable of sort *Nat* bound at level 1.

The LST is:

```
1 g?x:Nat
| 1 i(enable:exit(x))
| | 1 g!y
```

while the SST is:

```
1 g?Nat@1
| 1 i(enable:exit(Nat@1))
| | 1 g!Nat@1
```

Example:

```
choice x:Nat  []   g? y   [y lt x] ; stop
```

ing for this word. However it will usually be much more manageable than the original LST.

Our LOTOS interpreter [GHL] is able to systematically generate LSTs for a given process up to given maximum lengths and widths. Such trees show all possible execution sequences for the entity specified. When the maximum specified length along a path is exceeded, this is indicated by closing the path with a "continue". Paths exceeding the specified width, instead, are simply ignored, but the user is informed of this (of course, the user must be aware of the fact that, if some paths are ignored, some of the procedures discussed in this paper may yield incorrect results). A realistic example is shown in Section 2.1 of the Annex.

1.3. Overview of The Method

Unfortunately, the practical usefulness of LSTs is greatly reduced by the many unfeasible, redundant, or uninteresting paths that they contain. This is especially true for specifications written in the constraint-oriented style, where each action is subject to a number of logical constraints originating from different processes. Heuristics can be used in order to obtain more useful trees by detecting and eliminating some such paths.

- The first step is to obtain a *Significant Symbolic Tree* (or **SST** for short), where input variable values are represented by symbols derived from the variable's name. Some unfeasible paths or actions are detected and removed by using techniques similar to "symbolic evaluation".
- Loops in behavior are identified.
- Some non-significant internal events are detected and removed.
- All the previous steps are executed dynamically as the tree is generated by the interpreter. In a final step, the stored tree is scanned in order to eliminate some remaining redundant internal events or duplicate paths.

Needless to say, the resulting tree is by no means optimal, in any possible mean-

LOTOS" (i.e., LOTOS without data).

The slant of our paper is more pragmatic. We deal with full LOTOS specifications, and we obtain execution sequences from specifications by using an existing tool, i.e. our LOTOS interpreter.

As we shall see, the interpreter generates a great number of sequences that are either unfeasible, in the sense that they relate to logically impossible paths, or redundant, in the sense that they differ the ones from the others only by the placement of nonrelevant internal events. Of course, eliminating all impossible paths and taking out all nonrelevant internal events involves unsolvable problems. Therefore, these execution sequences are simplified by using various heuristics in order to make them useful for testing purposes.

This technique does not constitute (yet) a methodology for the derivation of test suites. Apart from the several possible improvements to be discussed later, the remaining steps, which are the selection of test sequences and the formulation of test sequences in a test specification language, must still be done by hand using ad hoc methods.

1.2. Labelled Symbolic Trees (LSTs)

By LOTOS semantics, given a behavior expression *B* one can find the set of actions *a* and the set of resulting behavior expressions *B'* such that: *B -a-> B'* , meaning that process *B* can execute action *a* and transform into *B'*. In other words, given a behavior expression, one can find its behavior tree. In the absence of an environment, actions that depend on guards or selection predicates which cannot be evaluated because this involves the knowledge of values that have to be provided by the environment must be listed, together with their guards. Such trees will be called Labelled Symbolic Trees *(***LST**s).

1 The influence of human interpretation (and interpretation errors) when producing test suites will be reduced. While the production of test suites from informal specifications requires the interpretation of the informal specification, this will not be needed in the proposed technique, since the interpretation will already have been fixed in the process of obtaining the formal specification. Test suites can be systematically derived from formal specifications and the need for human judgment will be limited to choosing what suites are the most revealing, and possibly to the interpretation of the final results.

2 Because the proposed techniques are expected to be automated to a large extent, their application can be expected to require less human effort than the application of current manual techniques. The establishment and maintenance of large libraries of test suites becomes then economically feasible.

3 Because all activities of test derivation will be based on automatic derivation of test suites from the same formal specification, the set of test suites obtained can be expected to be consistent. Current methodologies do not guarantee such consistency. This applies not only to mutual consistency between test suites obtained at different times and places, but also to internal consistency within a set of test suites generated together.

4 The automated techniques are expected to be more thorough than current ones in exercising subtle behavioral aspects.

5 Because LOTOS has a formally defined semantics, the meaning of LOTOS test suites can be more precisely determined than the meaning of test suites specified by informal or semi-formal methods.

Several results have already been reported on generating test suites from formal specifications. Some recent references are [DEM][FL][SBMS][UR][URS][BSS]. Eertink and Brinksma [EB] have developed an algorithm, based on a formal theory, for deriving "canonical testers" for a specification written in a restricted version of "pure

1. Introduction

1.1. Generating Test Suites from Formal Specifications

The general framework of our work is the family of Open Systems Interconnection (OSI) protocols and services. Such systems are currently being specified by informal methods (such as English prose), with the aid of semi-formal methods (such as state tables). Formal Description Techniques (FDTs), with formally specified semantics, have also been developed. One such technique is the language LOTOS, the Language of Temporal Ordering Specifications, which is used in this paper [ISO1][BB].

"Conformance testing" is an area of protocol development methodology that deals with the problem of testing that an implementation of a standard "conforms" to the standard specification. Current test methods for protocols and services usually derive test suites manually from informal descriptions or semi-formal ones. Since incompatible sets of test suites are likely to be derived from different interpretations of the standard, the test suites themselves are being standardized. Only test suites that are included in the standard are considered to reflect the "official" interpretation of the informal specification. This solution can only be considered as a temporary remedy, since then the test suites become the real formulation of the standard: the conformance of the implementation to the standard is no longer judged with respect to the text of the standard, but with respect to the test suites. In our view, it is the text of the standard that should dictate the validity of the test method, rather than vice-versa.

The methodology towards which this paper intends to be a contribution assumes instead that the behavior of the entity to be tested has already been specified precisely in LOTOS [ISO1][BB] and derives test suites automatically or semi-automatically from this specification.

Several advantages are expected to derive from this methodology.

Derivation of Useful Execution Trees

from LOTOS Specifications by Using an Interpreter

Renaud Guillemot and Luigi Logrippo

University of Ottawa
Computer Science Department
Ottawa, Ont., Canada K1N 9B4

ABSTRACT

A contribution towards the development of formal methodologies for testing protocol implementations is presented. We report on a system that is able to execute the specification of a protocol or service written in LOTOS and to derive an execution tree of the entity specified. Several heuristics are used in order to eliminate impossible or uninteresting execution paths. The tree obtained can then be used as a basis for the derivation of test suites.