Derivation of Test Cases for LAP-B from a LOTOS Specification

Djaffar Gueraichi and Luigi Logrippo

University of Ottawa
Computer Science Department
Protocols Research Group
Ottawa, Ont. Canada K1N 6N5
E-mail: LMLSL@UOTTAWA.BITNET

A semi-formal method for deriving test cases from specifications in LOTOS is described. The method is applied to the LAP-B standard, for which a complete LOTOS specification was developed. Execution trees are derived from the specification by using an interpreter, and test cases are obtained by inspection. The results of our experiment compare favorably with the ones obtained in an earlier work that used a state table description of LAP-B.

1. INTRODUCTION

In [KLPU] [Kan] a method for deriving conformance test suites for protocol implementations is discussed, which involves constructing the state transition tables for the protocol specification and then deriving from them test cases following three steps:

1) Initialization step, which brings the IUT (Implementation Under Test) to a specific test state

2) Evaluation step, which executes a single transition and checks the output of the IUT

3) Verification step, which verifies the state of the IUT after the transition.

This method was followed to obtain a test suite for the X.25 Level 2 (LAP-B) protocol [ISO5], suite that currently is in advanced state of standardization within ISO [ISO2][ISO3]. The method is based on a number of results in the area of protocol and finite-state machine testing, for which we refer to [KLPU][Kan][SL][SD].

In [GL] a method for deriving test cases from LOTOS [BB][ISO1] specifications by using an interpreter was proposed. Briefly, the method consists in constructing (possibly partial) symbolic execution trees of the specification. This can be done automatically, by using packages such as LOLA [QPF] or the University of Ottawa LOTOS interpreter [GHL] (see also [V]). These packages perform a (possibly partial) expansion [Mil] of the specification, by removing operators such as parallel composition and disable, and reducing the specification to a tree of alternatives. This tree explicitly indicates all possible execution sequences of the entity specified, at least up to some maximum length. Therefore, it can be used as a test tree by reversing the direction of the data flow (whenever the IUT accepts a value, the tester provides it, etc.). In this paper, we show how this method has been applied to obtain test cases for LAP-B that are comparable, and in fact occasionally better, than those obtained by [KLPU][Kan]. Since TTCN is a common language for the spec-

ification of test trees, the test cases obtained are written in TTCN (we should observe, however, that LOTOS itself appears to be adequate for the specification of test trees [Steen]).

This technique appears to be valuable for conformance testing, at least until such time as the more formal approaches being developed by other authors become available (see Section 7). It makes it possible to extract test cases directly from (possibly standardized) formal descriptions, eliminating or reducing the importance of the interpretation of the informally specified standard. The formal specification is more complete and precise than the state tables and, unlike the latter, allows full formal treatment of the *data* part. For example, we shall show that automatic or semi-automatic generation of frame values appear to be possible by using the information contained in selection predicates.

2. SYMBOLIC EXECUTION TREES FOR LOTOS PROCESSES

As mentioned above, the University of Ottawa LOTOS interpreter is capable of generating the symbolic execution tree of a process under interpretation, by using symbolic evaluation. This means that, in such trees, variable values are represented by variable names, with the addition of appropriate numerical suffixes to differentiate among different values for the same variable name. The construction algorithm is able to recognize cases where a previously encountered state (i.e., behavior expression) is found again, in which case the branch of the tree is closed by an *again*. In cases where the branch continues beyond a specified maximum length, it is closed by a *continue*. For example, consider the following process:

```
process disconnect[l,phl] : exit(Bool) :=
    (phl ? f:frame[Incorrect(f)];
     disconnect[l,phl])
    []
      l ? p:primitive[Is_ConReq(p)];
      phl ! SABM;
      (phl ? f : frame[Is_UA(f)];
        exit(true)
      []
        phl ? f : frame[Is_DM(f)];
        exit(false))
endproc
```

The tree generated is::

```
bh0 * 1 phl ?f:frame [Incorrect(f)] [13] ==> again bh0
      * 2 l ?p:primitive [Is_ConReq(p)] [16]
bh1 * | 1 phl !SABM:frame [17]
bh2 * | | 1 phl ?f:frame [Is_UA(f)] [18]
bh3 * | | | 1 exit !true:Bool ** EXIT SUCCEED ** [19]
      * | | 2 phl ?f:frame [Is_DM(f)] [21]
bh4 * | | | 1 exit !false:Bool ** EXIT SUCCEED ** [22]
```

It is possible that some branches of the tree involve predicates that can be evaluated to *false*. Such branches are pruned from the tree, yielding what we call the *Simplified Synchronization Tree* or SST [GL]. Note that value expressions containing symbolic values cannot be evaluated, thus predicates involving such expressions are assumed to be *true*, unless they can be evaluated to *false* without consideration of the symbolic values.

3. SPECIFICATION OF LAP-B IN LOTOS

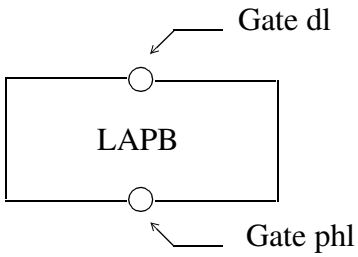
In order to perform this experiment, a full specification of the X.25 LAP-B was written. It includes all the options specified in the standard, for a total of some 2300 lines of LOTOS code [Gue].

In [VSV] it was shown how different specification styles can be used in LOTOS. Namely, that paper identified four major styles: the monolithic style, the constraint-oriented style [Tur], the state-oriented style, and the resource-oriented style. The constraint-oriented style is the one that is usually considered the most expressive and the most abstract, and accordingly, is the one that has most frequently been used in specifications of standards [LS][SAC][SS]. However, in [GL] it was shown that specifications written in the constraint-oriented style tend to yield unmanageably large execution trees, containing many infeasible paths (recall that predicates involving symbolic values are normally assumed to be *true*). Such paths could be pruned by using theorem-proving methods, however suitable tools are not available for LOTOS yet. Therefore, the technique described in this paper could not be used easily with the constraint-oriented style. Our specification is a mixture of the three other styles, each chosen according to the needs of executability and clarity.

The ISO/DIS 7776 [ISO5] specifies the X.25 level 2 LAPB interface operation as viewed by the DTE. The interface between LAPB and X.25 network layer is not defined in the ISO/DIS 7776. For this, primitives that allow communication between X.25 and LAPB have been defined. The LAPB can be accessed through a gate called *dl*. For example:

dl ? p: primitive [IS-CON-REQ(p)]

represents the possible occurrence of a connection request. The gate *phl* is the gate on which transmission and reception of frames is done by the DTE. This gate should be hidden, but in order to show more clearly the trees used in the test case generation, we made it visible. The figure below shows the gate configuration of the specification.



A point worth mentioning is the treatment of the timers. These are described as processes, which are composed in parallel with the processes that use them, and share with them a gate *t*. On this gate, operations such as *start, expired*, and *reset* are possible. A process can start or reset a timer, where a reset is modeled as disabling the timer. Similarly, a timer can perform an expiration, which is prefixed by an internal action and causes a disable in the main process. This treatment was found to be adequate for our problem.

Also, it is interesting to note that, although the specification makes little use of explicit states, the obtained execution trees show that there are seven states in LAP-B, and this corresponds to the results of [KLPU] [SHW].

4. TEST ARCHITECTURE

The test architecture we have assumed is the same as the one described in [KLPU], i.e. it is the one

of the remote test method. The tester interacts with the IUT Point of Control and Observation *L*, which corresponds to gate *phl* in the specification. This gate represents the DTE-DCE interface.

5. TEST SEQUENCE DERIVATION METHOD

The test sequence derivation method is based on the use of the SSTs and of the UIO (Unique Input-Output) method [SD] [SL]. In order to obtain manageable SSTs, the specification is divided in the three following phases:

-- Connection Phase.
-- Data Phase
-- Termination Phase.

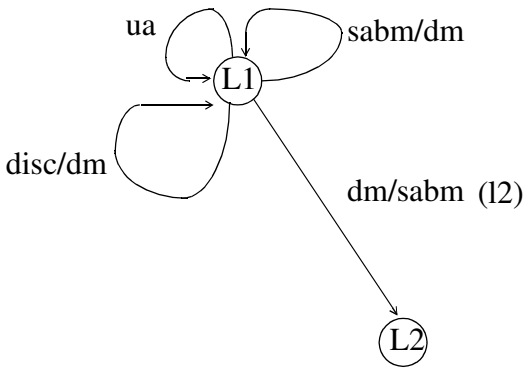
and for each phase the SSTs are generated. Note that the tester behaves the opposite way as the DTE. A receive (symbol *?*) in the obtained trees corresponds to a send (symbol *!*) for the tester; and a send in the obtained tree (symbol *!*) corresponds to a receive (symbol *?*) for the tester.

As mentioned above, a test case consists of three parts (1) initialization step (2) evaluation step and (3) verification step.

Initialization Step. The initialization step consists in finding an initialization path which brings the DTE to a given state. Consider the following example which shows the DTE at the Disconnected Phase. It is, of course, much simplified with respect to reality. The DTE Disconnected State can be represented by the following transition tree. This tree is a small part of the tree called *Disconnected* shown in the Appendix.

```
bh0 * 1 phl ?f : frame[is_disc(f)]
     * | 1 phl ! DM ==> again bh0
bh1 * 2 phl ?f : frame[is_dm(f)]
     * | 1 phl ! SABM ==> continue
bh2 * 3 phl ?f : frame[is_ua(f)] ==> again bh0
bh3 * 4 phl ?f : frame[is_sabm(f)]
     *  | 1 phl ! DM ==> again bh0
```

This transition tree can be represented by the following state diagram



The initialization path (*ipl1*) which brings the DTE to the Disconnected Phase (state *L1*) is not shown in this figure. The path *ipl1* is given by the Termination Phase part of the specification. It is added as a preamble to the test cases generated for state *L1*.

The initialization path (*ipl2*) which brings the DTE to state *L2* is obtained by following the edge (*l2*) which goes to state *L2*. The path (*ipl2*) is, therefore, obtained by concatenating path

ipl1 with edge *l2*:

```
(ipl2)      +ipl1
                L ! DM
                  L ? SABM
```

In general, given a current state, the transition tree for this state is obtained. Next, test cases for this state are generated (see evaluation step). Then the edges of the current transition tree which do not contain an *again* are explored, meaning that, probably, new states are reached. The initialization paths of the new states are obtained by concatenating the initialization path of the current state with those edges which lead to these states.

Evaluation Step. The evaluation step consists in obtaining the transition tree for the state following the initialization step, and then performing all the transitions given by this tree. A transition is a stimulus to which the DTE may or may not respond. It is constructed in the following way. For a reception of a frame *fs* in the transition tree, there is a transmission of the frame *fs* in the evaluation step. Successively, for a transmission of a frame *fr* in the transition tree, there is a reception of the frame *fr* in the evaluation step. The transmission of *fs* and the reception of *fr*, if any, represent the body of the test case.

If a frame *fr* is received after sending *fs*, the verification path of what follows the reception of *fr* is added to the body of the test case after reception of *fr*. If no reception follows the sending of *fs* and there is an exit to a next phase, the verification path of the next phase is added to the body of the test case. If no reception follows the sending of *fs* and there is no exit to a next phase, the verification path of next actions is added to the test case after the timer starts and stops (to make sure that nothing is received). An *otherwise* is added and a verdict *fail* is issued at any point where something different from what is expected to be received can occur. All this is summarized in the following:

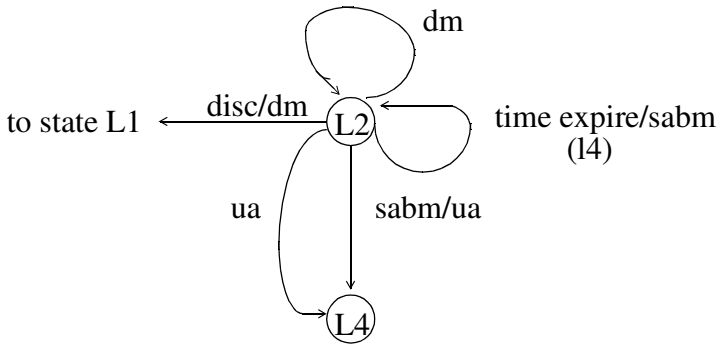
```
phl ! fs
 --if reception
        phl ? fr
          +verification path of what follows
 --else (no reception)
    --if exit to next phase
          + verification path of next phase
    --else (no exit to next phase)
           --start timer
              --time-out
                   + verification path of what follows
              --otherwise:                                    fail
 -- otherwise                                                 fail
 -- expiration of timer                                       fail
```

Consider the previous example with the transition tree of state *L2* as shown below. This tree can be considered as a small part of the tree called *Linksetup* shown in the Appendix.

```
bh0 * 1 phl ?f : frame[is_disc(f)]
     * | 1 phl ! DM ==> continue
bh1 * 2 phl ?f : frame[is_sabm(f)]
     * | 1 phl ! UA  ** EXIT SUCCEED **
bh2 * 3 phl ?f : frame[is_ua(f)]  ** EXIT SUCCEED **
```

```
bh3 * 4 phl ?f : frame[is_dm(f)] ==> again bh0
bh4 * 5 t ! expire : time
      * | 1 phl ! SABM ==> again bh0
```

This transition tree can be represented by the following state diagram. Edge (l4) expresses the fact that when a time-out occurs and the DTE does not receive anything, a SABM is sent and the DTE stays in state *L2* waiting for a response. Note that this condition is not tested.



Therefore the following four test cases are obtained:

```
(1) +ipl2
        L ! DISC
          L ? DM
            + verif_l1
          L ? Otherwise                    fail
          ?Elapse Td                       fail

(2) +ipl2
        L ! SABM
          L  ? UA
             + verif_l4
          L ? Otherwise                    fail
          ?Elapse Td                       fail

(3) +ipl2
       phl ! UA
            + verif_l4

(4)  +ipl2
         L ! DM
            (Start Timer)
               ?Timeout   Timer
                    + verif_l2
               L ? Otherwise               fail
```

The TTCN subtrees *verif_l1*, *verif_l2* and *verif_l4* are the verification paths of respectively states *L1, L2* and *L4*.

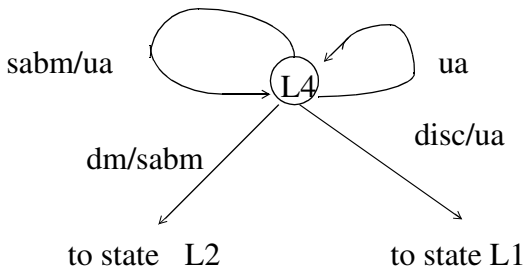
Verification Step. For the verification step, we must find a verification path. A verification path for a given state is obtained from the transition tree of that state using the UIO method, in other words we look for a path from the state which is unique to the corresponding transition tree. This path consists of either the transmission of a frame *fs* followed by the reception of a frame *fr*, or by

the reception of a frame *fr* before the timer expires. In this latter case, the tester starts its timer and does not send anything, in which case the DTE should send the frame *fr* after its timer expires. Therefore the tester should receive frame *fr* before its timer expires; otherwise the test case fails.

Consider again the previous example with the transition tree of state *L4* as shown below. This tree is a small part of the tree called *Dataphase* which, for reasons of space, cannot be shown in the paper (it is shown in [Gue]):

```
bh0 * 1 phl ?f : frame[is_disc(f)]
      * | 1 phl ! UA ==> continue
bh1 * 2 phl ?f : frame[is_sabm(f)]
      * | 1 phl ! UA  ==> again bh0
bh2 * 3 phl ?f : frame[is_ua(f)]  ==> again bh0
bh3 * 4 phl ?f : frame[is_dm(f)]
      * | 1 phl ! SABM ==> continue
```

This transition tree can be represented by the following state diagram:



We want to obtain the verification paths for states *L1*, *L2* and *L4*. Let us start by looking for the potential verification paths of each state. The potential verification paths of state *L1,* which are obtained from the transition tree of state *L1*, are:

```
(pvp1.1)  L ! DISC
                L ? DM                  pass
                L ? Otherwise           fail
                ?Elapse Td              fail

(pvp1.2)  L ! SABM
                L ? DM                  pass
                L ? Otherwise           fail
                ?Elapse  Td             fail

(pvp1.3)  L ! DM
                L ? SABM                pass
                L ? Otherwise           fail
                ?Elapse  Td             fail

(pvp1.4)  L ! UA
               (Start Timer)
                  ?Timeout  Timer       pass
                  L ? Otherwise         fail
```

The potential verification paths for state *L2* are:

(pvp2.1) L ! DISC
L ? DM pass
L ? Otherwise fail
?Elapse Td fail

(pvp2.2) L ! SABM
L ? UA pass
L ? Otherwise fail
?Elapse Td fail

(pvp2.3) L ! UA
(Start Timer)
?Timeout Timer pass
L ? Otherwise fail

(pvp2.4) L ! DM
(Start Timer)
?Timeout Timer pass
L ? Otherwise fail

(pvp2.5) (Start Timer)
L ? SABM pass
L ? Otherwise fail
?Timeout Timer fail

The potential verification paths for state *L4* are:

(pvp4.1) L ! DISC
L ? UA pass
L ? Otherwise fail
?Elapse Td fail

(pvp4.2) L ! SABM
L ? UA pass
L ? Otherwise fail
?Elapse Td fail

(pvp4.3) L ! DM
L ? SABM pass
L ? Otherwise fail
?Elapse Td fail

(pvp4.4) L ! UA
(Start Timer)
?Timeout Timer pass
L ? Otherwise fail

The potential verification paths which are common to some states are removed from the potential verification paths of those states. Paths (*pvp1.1*) and (*pvp2.1*) are identical. Paths (*pvp1.3*) and (*pvp4.3*) are identical. Paths (*pvp2.2)* and (*pvp4.2*) are identical. Paths (*pvp1.4*), (*pvp 2.3*) and (*pvp4.4*) are identical. For state *L1* the only potential verification path that remains is

(*pvp1.2*). For state *L2* the only potential verification paths that remain are (*pvp2.4*) and (*pvp2.5*). For state *L4* the only potential verification path that remains is (*pvp4.1*). Now these potential verification paths must be compared with all potential verification paths of all other states, including those yet to be considered, so the same process must be repeated with respect to all these other paths.

Of course, such unique I/O sequences are not guaranteed to exist in general. For LAP-B, however, we were always able to find them.

Since (*pvp4.1*) was found to be the verification path for state *L4*, a complete test case (2) for state *L2* (see Evaluation Step) is:

```
+ipl2
    L ! SABM
        L ? UA
            +(pvp4.1)
        L ? Otherwise                fail
        ?Elapse Td                   fail
```

Resolution of Selection Predicates in SSTs (The Treatment of Data). It has already been mentioned that to the reception of a frame by the IUT, shown in the generated SSTs by the symbol *?*, corresponds the transmission of a frame by the tester. Selection predicates may be associated with this reception. Frame value(s) must be generated from these selection predicates, such that the predicates are true. These frames are the ones to be sent by the tester, and to each frame value corresponds a test case. For example, consider the following frame reception

phl ?f : frame[is-disc(f)]

the only frame value which can be generated from the selection predicate *[is-disc(f)]* is a DISC frame. In reality, more complex selection predicates that admit more that one frame value appear. Assume that the following reception is shown in a tree:

phl ?f : frame[is-disc(f) or (pf(f) eq 1)]

The selection predicate of this reception, *[is-disc(f) or (pf(f) eq 1)]* means that a DISC frame or any frame for which the field *pf* is equal to *1* are acceptable. Suitable instances of such frames must be generated for testing purposes. The selection predicate shown in the tree can also be specified as a combination of predicates. The following example illustrates this:

phl ?f : frame[Incorrect(f)]

where the predicate *Incorrect* is defined in the specification as:

Incorrect(f) = Not(Is-valid(f)) or Not(Not-abort(f))

Again, in order to specify the test cases one must see how predicates *Is-valid* and *Not-abort* are defined in the specification, and frame values which satisfy either predicate *Not(Is-valid(f))* or predicate *Not(Not-abort(f))* must be generated.

The frame generation was done manually. However, it would be possible to automate it to a certain extent, for example by using logic programming techniques.

6. COMPARISON

Following is a brief comparison between our method and results and those of [Kan][KLPU]. The method is indeed similar, however LOTOS enables a much more precise description of protocols than state tables, since it allows to specify not only the control structure, but also the data and related constraints. Furthermore, while the test cases of [Kan][KLPU] were obtained manually from a state table, we were able to use the tree-building facilities of our LOTOS interpreter.

The results are also similar, at least no differences were found in the test cases obtained for the initialization and evaluation steps. In the verification steps, however, we were able to find, in some cases, what we believe to be better verification sequences, as documented in [Gue]. Unfortunately, space does not allow to discuss this point here.

7. CONCLUSIONS

We have presented a semi-formal method for deriving test cases from LOTOS specifications, and we have shown how the method has been applied in order to obtain test cases for LAP-B. The results obtained have been shown to compare favorably with those of a similar semi-formal method based on the use of state tables.

Our method is semi-formal because it has no claim of completeness and depends for its use on the skill of the test specifier in finding relevant paths. Furthermore, it depends on the use (or avoidance) of certain LOTOS specifications styles. Based on LOTOS theory, a formal method is being developed, for deriving test cases from specifications [Brink1][Brink2][E][L][W]. So far, however, this method is of limited practical value because it only applies to a small subset of LOTOS. In [LS] a method which appears somewhat related to ours is presented, and is applied to the Alternating Bit Protocol. Hopefully, with further study it will become clear how these different approaches can be integrated and it will be possible to find techniques that are more formal and more automated. We believe that our method is valuable at least until such time as this research has developed to the point where it is able to handle specifications of complexity comparable to the one discussed here.

ACKNOWLEDGMENTS. Work reported here was partially funded by a number of sources: the Natural Sciences and Engineering Research Council, the University Research Incentive Fund, the Department of Communications, the Telecommunications Research Institute of Ontario, Bell-Northern Research, and the Algerian government. Several colleagues of the Protocols Research Group of the University of Ottawa provided very valuable suggestions and advice. In particular, we are grateful to R.L. Probert and H. Ural for valuable comments on earlier versions of this paper.

Appendix: Some Trees Used in Test Case Generation

Disconnected Tree

```
bh0  * 1 phl ?[sframe,sframe]eframe  (1)
          [or(and(correct(sframe),and(eq(pf(sframe),$1),eq(adrs(sframe),from))),is_disc(get_f(sframe)))]
          [not(or(and(eq(tmode,basic),is_sabm(get_f(sframe))),and(eq(tmode,extended),is_sabme(get_f(sframe)))))]  [619,622]
bh1  * | 1 i (enable: exit !pf(sframe):Bit) [620,623]
bh2  * | | 1 phl !m(make_dm(flagging,from,ctl_uframe(b),fc,flagging),noinfor,correct,notless,noabort):eframe [624]
bh3  * | | | 1 i (enable: exit) [626] ==> again bh0
     * 2 phl ?[sframe,sframe]eframe(2)
          [and(not(or(and(correct(sframe),and(eq(pf(sframe),$1),eq(adrs(sframe),from))),is_disc(get_f(sframe)))),
          not(and(is_dm(get_f(sframe)),eq(pf(sframe),$0))))]
          [not(or(and(eq(tmode,basic),is_sabm(get_f(sframe))),and(eq(tmode,extended),is_sabme(get_f(sframe)))))]  [629,633]
```

```
bh4  * | 1 i (enable: exit) [631,634] ==> again bh0
       * 3 dl ?p:primitive(3)
          [is_dl_est_req(p)]  [639]
bh5  * | 1 [eq(tmode,basic)] i (hiding: [lt(init,n2)] p !$1:Nat) [1096,1150]
bh6  * || 1 phl !f:eframe [1097,1151]
bh7  * ||| 1 i (hiding: t !start:time) [1078,547] ==> continue
       * | 2 [eq(tmode,basic)] i (hiding: [eq(init,n2)] p !$0:Bit) [1108,1153]
bh8  * || 1 exit !false:Bool !false:Bool ** EXIT SUCCEED ** [1109,1154]
       * | 3 [eq(tmode,extended)] i (hiding: [lt(init,n2)] p !$1:Nat) [1096,1150]
bh9  * || 1 phl !f:eframe [1097,1151]
bh10* ||| 1 i (hiding: t !start:time) [1078,547] ==> continue
       * | 4 [eq(tmode,extended)] i (hiding: [eq(init,n2)] p !$0:Bit) [1108,1153] ==> again bh8
       * | 5 i (hiding: [lt(init,n2)] p !$1:Nat) [1096,1150]
bh11 * || 1 phl !f:eframe [1097,1151]
bh12 * ||| 1 i (hiding: t !start:time) [1078,547] ==> continue
       * | 6 i (hiding: [eq(init,n2)] p !$0:Bit) [1128,1166] ==> again bh8
       * | 7 [senddmoption] i (hiding: [lt(init,n2)] p !$1:Nat) [682,719]
bh13 * || 1 phl !m(make_dm(flagging,to,ctl_uframe($0),fc,flagging),noinfor,correct,notless,noabort):eframe [683,720]
bh14 * ||| 1 i (hiding: t !start:time) [684,536] ==> continue
       * | 8 i (hiding: [eq(init,n2)] p !$0:Bit) [695,722] ==> again bh8
       * 4 phl ?sframe:eframe(4)
          [and(or(and(is_sabm(get_f(sframe)),eq(tmode,basic)),and(is_sabme(get_f(sframe)),eq(tmode,extended))),
           correct(sframe))]  [596]
bh15 * | 1 i [598]
       * || 1 phl !m(make_ua(flagging,from,ctl_uframe(pf(sframe)),fc,flagging),noinfor,correct,notless,noabort):eframe [598]
bh16 * ||| 1 exit !true:Bool !false:Bool ** EXIT SUCCEED ** [598]
       * | 2 i [602]
       * || 1 phl !m(make_dm(flagging,from,ctl_uframe(pf(sframe)),fc,flagging),noinfor,correct,notless,noabort):eframe [602]
bh17 * ||| 1 exit !false:Bool !false:Bool ** EXIT SUCCEED ** [603]
       * 5 phl ?sframe:eframe(5)
          [and(correct(sframe),and(is_dm(get_f(sframe)),eq(pf(sframe),$0)))]  [607]
       * | 1 i [609]
bh18 * || 1 i (hiding: [lt(init,n2)] p !$1:Nat) [1096,1150]
bh19 * ||| 1 phl !f:eframe [1097,1151] ==> again bh7
       * || 1 i (hiding: [eq(init,n2)] p !$0:Bit) [1128,1166] ==> again bh8
       * | 2 i [613]
       * || 1 phl !m(make_disc(flagging,to,ctl_uframe(pf(sframe)),fc,flagging),noinfor,correct,notless,noabort):
            eframe [613] ==> again bh17
       * 6 phl ?f:eframe(6)
            [not(correct(f))]  [830] ==> again bh3
```

Linksetup Tree

```
bh0  * 1 i (hiding: [lt(init,n2)] p !Succ(0):Nat) [1096,1150]
bh1  * | 1 phl !m(make_sabm(flagging,b,ctl_uframe($1),fc,flagging),noinfor,correct,notless,noabort):eframe [1097,1151]
bh2  * || 1 i (hiding: t !start:time) [1098,547]
bh3  * ||| 1 i (specified explicitly) [548](1)
bh4  * |||| 1 i (hiding: t !expired:time) [1100,549]
bh5  * ||||| 1 i (enable: exit) [1100]
bh6  * |||||| 1 i (hiding: [lt(init,n2)] p !Succ(0):Nat) [1096,906,931,956,992,1027]
bh7  * ||||||| 1 phl !m(make_sabm(flagging,b,ctl_uframe($1),fc,flagging),noinfor,correct,notless,noabort):
                    eframe [1097,907,932,957,993,1028] ==> again bh2
       * |||||| 2 i (hiding: [eq(init,n2)] p !0:Bit) [1108,1153]
bh8  * ||||||| 1 exit !false:Bool !false:Bool  !$f1  ** EXIT SUCCEED ** [1109,1154,550]
bh9  * ||| 2 phl ?[fram,sframe,f,f,f,f]eframe(2)
        [and(correct(sframe),or(is-sabm(get-f(sframe)),or(or(is-disc(get-f(sframe)),and(is-ua(get-f(sframe)),
             eq(pf(sframe),1))),and(is-dm(get-f(sframe)),eq(pf(sframe),1)))))]
        [and(correct(f),and(is-ua(get-f(f)),eq(pf(f),1)))]
        [not(and(correct(f),and(is-dm(get-f(f)),eq(pf(f),1))))]
        [not(and(correct(f),or(is-sabm(get-f(f)),is-sabme(get-f(f)))))]
        [not(and(correct(f),is-disc(get-f(f))))]  [1127,882,921,938,979,1000]
       * ||||| 1 i (hiding: pp !0:Nat) [1113,890,922,941,983,1003]
bh10 * |||||| 1 i (hiding: t !reset:time) [1114,552]
bh11 * ||||||| 1 exit !true:Bool !false:Bool !$f1:eframe  ** EXIT SUCCEED ** [1115,891,917,942,984,1004,553]
```

```
      * | | | 3 phl ?[fram,sframe,f,f,f,f]eframe(3)
          [and(correct(sframe),or(is-sabm(get-f(sframe)),or(or(is-disc(get-f(sframe)),and(is-ua(get-f(sframe)),
           eq(pf(sframe),1))),and(is-dm(get-f(sframe)),eq(pf(sframe),1)))))]
          [not(and(correct(f),and(is-ua(get-f(f)),eq(pf(f),1))))]
          [and(correct(f),and(is-dm(get-f(f)),eq(pf(f),1)))]
          [not(and(correct(f),or(is-sabm(get-f(f)),is-sabme(get-f(f)))))]
          [not(and(correct(f),is-disc(get-f(f))))]  [1127,882,913,945,979,1000]
      * | | | | | 1 i (hiding: pp !0:Nat) [1113,890,916,947,983,1003]
bh12 * | | | | | | 1 i (hiding: t !reset:time) [1114,552]
bh13 * | | | | | | | 1 exit !false:Bool !false:Bool !$f6:eframe  ** EXIT SUCCEED ** [1115,891,917,948,984,1004,553]
      * | | | 4 phl ?[fram,sframe,f,f,f,f]eframe(4)
          [and(correct(sframe),or(is-sabm(get-f(sframe)),or(or(is-disc(get-f(sframe)),and(is-ua(get-f(sframe)),
          eq(pf(sframe),1))),and(is-dm(get-f(sframe)),eq(pf(sframe),1)))))]
          [not(and(correct(f),and(is-ua(get-f(f)),eq(pf(f),1))))]
          [not(and(correct(f),and(is-dm(get-f(f)),eq(pf(f),1))))]
          [not(and(correct(f),or(is-sabm(get-f(f)),is-sabme(get-f(f)))))]
          [and(correct(f),is-disc(get-f(f)))]  [1127,882,913,938,979,1007]

      * | | | | 1 i (enable: exit !false:Bool !m(make_disc(flagging,b,ctl_uframe($1),fc,flagging),noinfor,correct,notless,noabort)
                 eframe) [1008]
bh14  * | | | | | 1 i (hiding: pp !$1:Nat) [1118,890,916,941,983,1011]
      * | | | | | | 1 i (hiding: t !reset:time) [1119,552]
      * | | | | | | | 1 i (hiding: p !$0:Nat) [1120,900,925,950,986,1012]
bh15  * | | | | | | | | 1 phl !m(make_dm(flagging,a,ctl_uframe(1),fc,flagging),noinfor,correct,notless,noabort)
                  :eframe [1121,901,926,951,987,1013]
       * | | | | | | | | | 1 exit !false:Bool !false:Bool !$f8  ** EXIT SUCCEED ** [1122,902,927,952,988,1014,550]
       * | | | 5 phl ?[fram,sframe,f,f,f,f]eframe(5)
          [and(correct(sframe),or(is-sabm(get-f(sframe)),or(or(is-disc(get-f(sframe)),and(is-ua(get-f(sframe)),
           eq(pf(sframe),1))),and(is-dm(get-f(sframe)),eq(pf(sframe),1)))))]
          [not(and(correct(f),and(is-ua(get-f(f)),eq(pf(f),1))))]
          [not(and(correct(f),and(is-dm(get-f(f)),eq(pf(f),1))))]
          [and(correct(f),or(is-sabm(get-f(f)),is-sabme(get-f(f))))]
          [not(and(correct(f),is-disc(get-f(f))))]  [1127,882,913,938,964,1000]
       * | | | | 1 i (enable: exit !true:Bool  !m(make_sabm(flagging,b,ctl_uframe($1),fc,flagging),noinfor,correct,notless,noabort)
            :eframe) [965]
       * | | | | | 1 i (hiding: [collis(m(make_sabm(flagging,b,ctl_uframe($1),fc,flagging),noinfor,correct,notless,noabort),
                  m(make_sabm(flagging,b,ctl_uframe($1),fc,flagging),noinfor,correct,notless,noabort))]
                  pp !$1:Nat) [1118,890,916,941,1035,1003] ==> continue
bh16   * | | | | | | 1 i (hiding: t !reset:time) [1119,552]
bh17   * | | | | | | | 1 i (hiding: p !$0:Nat) [1120,900,925,950,1036,1021]
bh18   * | | | | | | | | 1 phl !m(make_ua(flagging,a,ctl_uframe(1),fc,flagging),noinfor,correct,notless,noabort) :
                    eframe [1121,901,926,951,1037,1022] ==> continue
       * | | | 6 phl ?[fram,f,f,sframe,f,f]eframe(6)
          [not(and(correct(f),or(is-sabm(get-f(f)),is-sabme(get-f(f)))))]
          [not(and(correct(f),is-disc(get-f(f))))]
          [not(or(and(correct(sframe),is-sabm(get-f(sframe))),or(is-disc(get-f(sframe)),or(and(is-ua(get-f(sframe)),
           eq(pf(sframe),1)),and(is-dm(get-f(sframe)),eq(pf(sframe),1))))))]
          [not(and(correct(f),and(is-ua(get-f(f)),eq(pf(f),1))))]
          [not(and(correct(f),and(is-dm(get-f(f)),eq(pf(f),1))))]  [1127,895,913,938,979,1000] ==> again bh9
```

References

[BB] E. Brinksma, T. Bolognesi. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN systems 14* (1987) 25-59.

[Brink1] E. Brinksma. A theory for the derivation of tests. In: S. Aggarwal and K. Sabnani (eds.) *Protocol Specification, Testing, and Verification VIII,* North-Holland 1988 19-31.

[Brink2] E. Brinksma. On the existence of canonical testers. Technical report, University of Twente, January 1987.

[E] E.H. Eertink. The implementation of a test derivation algorithm. Universiteit Twente, Faculteit der Informatica, Memorandum INF-87-36 (1987).

[GHL] R. Guillemont, M. Haj-Hussein, L. Logrippo. Executing large LOTOS specifications. In: S. Aggarwal and K. Sabnani (eds.) *Protocol Specification, Testing, and Verification VIII,* North-Holland 1988, 399-410.

[GL] R. Guillemot, L. Logrippo. Derivation of useful execution trees from LOTOS specifications by using an interpreter. In: K.J. Turner (ed.) *Formal Description Techniques.* North-Holland 1988, 311-327

[Gue] D. Gueraichi. Derivation of test cases for LAP-B from a formal specification in LOTOS. MCS Thesis, University of Ottawa, June 1989.

[ISO1] International Organization for Standardization, Information Processing Systems, Open Systems Interconnection. LOTOS - A formal description technique based on temporal ordering of observational behaviour (ISO IS 8807), 1988.

[ISO2] International Organization for Standardization, Telecommunications and Information Exchange Between Systems. X.25-DTE conformance testing: Revised text for Data Link Layer test suite for DP 8882 Part 2. ISO/JTC1/SC6/WG1, April 1988.

[ISO3] International Organization for Standardization, Telecommunications and Information Exchange Between Systems. X.25-DTE conformance testing: Revised text for Data Link Layer test suite for DP 8882 Part 2, ISO/IEC TC1/SC 6/WG 1, January 1989.

[ISO4] International Organization for Standardization, Information Retrieval, Transfer and Management for OSI. The Tree and Tabular Combined Notation (TTCN) (DP9646-3), ISO/IEC JTC 1/SC 21 N3077 Sydney Meeting, February 1989.

[ISO5] International Organization for Standardization. Description of the 1984 X.25 LAPB-Compatible DTE Data Link procedures" (ISO 7776), ISO TC 97/ SC6, 1984.

[KLPU] B. Kanungo, L. Lamont, R.L. Probert, and Ural, H. A useful FSM representation for test suite design and development. In: B. Sarikaya and G.v. Bochmann (eds) *Protocol Specification, Testing, and Verification VI.* North-Holland, 1987.

[Kan] B. Kanungo. Specification and design of an OSI standard test suite for X.25 DTEs. MCS Thesis, University of Ottawa, 1987.

[GL] R. Guillemot, L. Logrippo. Derivation of useful execution trees from LOTOS specifications by using an interpreter. In: K.J. Turner (ed.) *Formal Description Techniques.* North-Holland 1988, 311-327

[L] R. Langerak. A Testing theory for LOTOS using deadlock detection.To appear in: E. Brinksma, G. Scollo, and C.A. Vissers (Eds.) *Protocol Specification, Testing, and Verification, IX.* North-Holland.

[LS] S. Lu, B. Sarikaya. A LOTOS-based test design methodology. Concordia University Technical Report, December 1988.

[LSC] J. Van de Langemaat, G. Scollo. On the use of LOTOS for the formal description of a

Transport Protocol. In: K.J. Turner (ed.) *Formal Description Techniques.* North-Holland 1988, 247-261.

[Mil] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science Vol. 92, Springer-Verlag 1980.

[QPF] J. Quemada, S.Pavon, A. Fernandez. Transforming LOTOS specifications with LOLA. The parameterized expansion. In: K.J. Turner (ed.) *Formal Description Techniques.* North-Holland 1988, 45-54.

[SD] K. Sabnani, A.T. Dahbura. A protocol test generation procedure. *Computer Networks 15* (1988) 285-297.

[SHW] M.H. Sherif, G.L. Hoover, R.P. Wiederhold. X.25 conformance testing - A tutorial. *IEEE Communications 24* (1986) 16-27.

[SL] D. Sidhu, T.K. Leung. Formal methods for protocols testing: a detailed study. Department of Computer Science, Iowa University, December 1986.

[SS] G. Scollo, M. van Sinderen. On the architectural design of the formal specification of the Session Standards in LOTOS. In B. Sarikaya, and G.V. Bochmann (eds) *Protocol Specification, Testing, and Verification VI,* North-Holland 1987, 3-14.

[Steen] C. Steenbergen. Conformance testing of OSI systems. Master Thesis, University of Twente, August 1986.

[Tur] K. Turner. Constraint-oriented style in LOTOS. In: Proc. of the British Computer Society Workshop on Formal Methods in Standards. Didcot, 1988.

[V] P. van Eijk. Software tools for the specification language LOTOS. PhD Thesis, Universiteit Twente, 1988.

[W] C.D. Wezeman The CO-OP method for compositional derivation of conformance testers. To appear in: E. Brinksma, G. Scollo, and C.A. Vissers (Eds.) *Protocol Specification, Testing, and Verification, IX.* North-Holland.

[VSS] C.A. Vissers, G. Scollo, M.v. Sinderen. Architecture and specification styles in formal descriptions of distributed systems". In: S. Aggarwal and K. Sabnani (eds.) *Protocol Specification, Testing, and Verification VIII,* North-Holland 1988, 189-204.