Chapter 1. Introduction

1.1 Background

As interconnection of computer networks has become more common, the necessity to produce open systems, systems which are capable of interworking with a vast variety of other systems, has become a requirement. This has led the International Organization for Standardization (**ISO**) to work on the development of international standards for Open Systems Interconnection (**OSI**).

In order to communicate meaningfully, all open systems must conform to these standards. This conformance implies that different implementors hold common interpretation of the standards, which requires the standards to be specified in notations that do not leave any space for ambiguity or impreciseness. This is a difficult task to achieve if the standards are specified in informal methods, consisting of natural languages or various semi-formal notations, which has been the case so far.

To provide for common interpretation of the standards, ISO has developed two complementary approaches: the use of FDT for specifying formally protocols and services, and precise methods for specifying and executing tests of conformance.

1.2 Formal Description Techniques

The main objectives of FDT's are to allow the production of OSI standards specifications that are unambiguous, precise and complete and to provide a formally well-defined basis for verification and validation of the standards as well as for the conformance testing of their implementation [BB].

Formalization of protocols specifications is important, if not essential, for improving the productivity of conformance test suite design and development. Applications of formalized specifications are [BGPU]:

-- possibility of automating or semi-automating the generation of test cases.

-- validation of the specification.

-- validation of a test suite against the formalized specification.

-- possibility of generating an implementation from the formal specification.

The original means of specifying protocols was with Finite State Machines (FSM). Then, languages based on Extended Finite State Machine (EFSM), like ESTELLE, have appeared. More recently, LOTOS has been proposed as a language for specifying OSI protocols and services.

LOTOS has a different philosophy from EFSM-based languages. These are useful for describing reference implementations of OSI standards, while LOTOS, because of its abstractness, allows one to produce implementation-independent specifications.

The strong mathematical basis of LOTOS allows it to have a precise formal semantics. An important feature of the semantics of LOTOS is that, even if it allows executability, it admits a 'process algebra' making possible the development of proof methods. However, such methods are not practical for specifications of realistic size, so this area requires much additional work.

The communication is synchronous in LOTOS, while it is asynchronous in languages like ESTELLE. As far as parallelism is concerned, LOTOS is very powerful in its capability to describe process interaction, while for EFSM based languages process parallelism is very limited.

Although it is a 'young' language, LOTOS posses tools which allow checking of syntax and semantics or carrying out simulation. Also, tools (validation tools) are available for EFSM-based languages. Finally, note that the expressive power of LOTOS makes it also suitable for concurrent, distributed systems.

1.3 Test of Conformance

The purpose of conformance testing is to increase the probability that different implementations are able to interwork. This is achieved by verifying them by means of a protocol test suite, thereby increasing the confidence that each implementation conforms to the protocol specification. Confidence in conformance to a protocol specification is particularly important when equipment supplied by different vendors is required to interwork [ISO9].

However, it should be noted that the complexity of most protocols makes exhaustive testing impractical on both technical and economical points of view. Also, testing cannot guarantee conformance to a specification since it detects the presence of errors rather than their absence.

Currently, test methods for protocols usually derive test suites manually from informal or semi-formal descriptions. In this way, incompatible sets of test suites could possibly be derived from different interpretations of the standard, if the organizations involved in test generation work separately. However, collaboration that occurs between major corporations and standard bodies tends to reduce or eliminate this possibility. Also, a number of test suites have been proposed as international standards.

1.4 Test Case Derivation from Formal Specifications

It would be more appropriate to generate test cases from protocols specified in FDT notations since these provide a means for protocol description in a precise and unambiguous manner. But this assumes the existence of methods which allow the derivation of test cases from formal specifications. Also, tools which implement these methods would be very helpful. For FSM, there are methods and tools allowing the derivation of test cases for control flow (see 4.2). For the EFSM based languages, there are methods and tools allowing the derivation of test cases for data flow information [Ural].

On the other hand, it is quite understandable that there is no general method for generating test cases for LOTOS, since LOTOS is rather a 'new' language. There exists, however, a method, with a theoretical foundation, that permits to derive test cases for the control flow, for a limited subset of LOTOS (see 4.3).

However, it should be noted that, at the current state of the knowledge, it is difficult to foresee the production of tools capable of generating test cases in a fully automated way. For example, one can expect that the selection of data values has to be done by a testing expert, on the basis of his experience.

1.5 Objective of the Thesis

As mentioned previously, current standard test suites are derived manually from informal or semi-formal specifications. Since informal or semi-formal notation could lead to an ambiguity or imprecision, incorrect test cases could be generated. Therefore it is appropriate to validate the test cases with respect to the corresponding formal specification.

One can argue that standard test suites were obtained by experts who spent a lot of time in doing this. Therefore, it is possible to have confidence in these suites. Nevertheless, it is still useful, in our view, to check the consistency of a test suite with respect to a formal specification.

This can be done in two ways: either generate test cases from the formal specification and then compare the obtained test cases with the given test cases; or check if the given test cases are accepted by the formal specification. One of the advantage of the first alternative is the possibility of obtaining some new interesting test cases.

The objective of this thesis was to validate test cases derived from a DTE LAPB specified by a FSM with respect to a LOTOS specification of the DTE LAPB. As mentioned in section 4.4, the test cases were derived from a LAPB FSM by B. Kanungo [Kan] and his work was further refined in [ISO4] and [ISO5].

In our work, the first step consisted of writing a specification of the DTE LAPB in LOTOS. The second step consisted of validating the given test suite. The method used consisted of generating test cases from the formal specification in LOTOS of the LAPB, and then the obtained test cases were compared with the given test cases. The derivation of test cases from the LOTOS specification is done by a semi-automatic method based on the use of the LOTOS interpreter.

Chapter 2. The LOTOS Language

2.1 Introduction to LOTOS

LOTOS is an FDT, developed within ISO, for the formal specification of data communications systems; however it is generally applicable to distributed, concurrent, information processing systems. Developed in the early eighties, it is now an ISO standard (ISO 8807).

LOTOS consists of two components: the first component, based on Milner's Calculus of Communicating Systems (CCS) and on Hoare's Communicating Sequential Processes (CSP), describes the 'control' or 'behaviour' part; the second component, based on ACT ONE abstract data type formalism, describes the 'data' part. LOTOS has been designed as an executable specification language even if executability was not the primary goal of its design. Therefore tools which allow to check the static and dynamic semantics of LOTOS specifications have appeared. This has the advantage of allowing testing to start at the specification stage.

2.2 LOTOS Concepts

A complete tutorial of LOTOS can be found in [BB]; however a brief overview of its main concepts is presented here.

A LOTOS specification consists of a data part and a behaviour part. The data part, using the abstract data typing language ACT ONE, describes the data structure manipulated by the behaviour part. An abstract data type is an implementation independent representation of structured data; it is a combination of *sorts*, *operations* and *equations*. The *sorts* are names of sets of elements, *operations* are functions on the elements, and *equations* describe axioms of the type. From the information given by the data type specification syntactically correct terms called value expressions are built. The data part will not be discussed in this thesis.

The behaviour part is a combination of processes using LOTOS operators. A process is an entity able to perform internal unobservable actions, and to interact with its environment by means of interactions via interaction points called 'gates'. The atomic form of interaction is an *event*. An *event* is a unit of synchronization that may exist between two processes that can perform that event.

A process is written using LOTOS operators. Taking the notation below:

-- *P*: process name.

-- *e*: event.

-- *A, B*: process names or behaviour expressions.

-- *C*: condition evaluates to true or false.

-- $g, g_1, g_2 ..., g_n$: gate names.

-- $v, v_1, v_2 ..., v_n$: value expressions.

-- $t, t_1, t_2 ..., t_n$: sort types.

-- $x, x_1, x_2 ..., x_n$: variable names.

-- *op:* one of the three parallel operators |||, || or |[]|.

-- *{ }*: what is between the symbol '{' and the symbol '}' is optional.

-- $y_1 \backslash y_2$: parameter y_1 replaces parameter y_2.

the different operators are briefly described in the following:

-- *stop*: deadlock.

-- *e; B*: event e is performed then behaviour B is done (action prefix).

-- *exit* $\{v_1,...,v_n\}$: successful termination and $v_1...,v_n$ are passed to the enabled behaviour.

-- $P[g_1...,g_n](v_1...v_n)$: process instantiation.

-- *A [] B*: behaves like A or B (choice).

-- *[C] -> B*: behaves like B if C is true.

-- *A ||| B*: both A and B progress in parallel (interleaving).

-- *A || B*: A and B must synchronize among themselves on all gates (full synchronization).

-- $A\ |[g_1,...,g_n]|\ B$: A and B have to synchronize on gates $g_1,...,g_n$ (selected synchronization).

-- *hide* $g_1,...,g_n$ *in B*: any action offered by B on gates $g_1,...,g_n$ will be hidden from the environment (hiding).

-- *A [> B*: B can disable A and start execution at any time during the life of A, unless A terminates successfully.

-- *A >> B*: when A is successfully terminated, the execution of B is enabled (enable).

-- *let* $x_1{:}t_1{=}v_1,...,x_n{:}t_n{=}v_n$ *in B*: replace the occurrence of $x_1,...,x_n$ by $v_1,...,v_n$ in B (local definition).

-- *choice g in [g_1,...,g_n] in B*: this is equivalent to B[g_1/g] []... [] B[g_n/g].

-- *choice x:t [] B*: this is equivalent to [v_1/x] B []... [] [v_n/x] B where v_1,...,v_n are value expressions of sort t offered by the environment.

-- *par g in [g_1,...,g_n] op B*: this is equivalent to B[g_1/g] op... op B[g_n/g].

Although for reasons of space we do not discuss the data part, we give here some common abbreviations that will be used in the thesis:

-- *eq*: equal.

-- *ne*: not equal.

-- *lt*: less than.

-- *le*: less or equal

.-- --+: insertion of an element into a queue.

-- $X: abbreviation allowed by the interpreter in order to enter the predefined constant X.

2.3 The Use of the LOTOS Interpreter

2.3.1 Introduction

The use of an interpreter for specification languages, such as LOTOS, is very useful even if unlike implementation languages, specification languages do not have to be executable. Actually LOTOS philosophy involves executability although it is possible to obtain LOTOS constructs that are difficult to execute (such constructs would have to be modified for the specification to be executable).

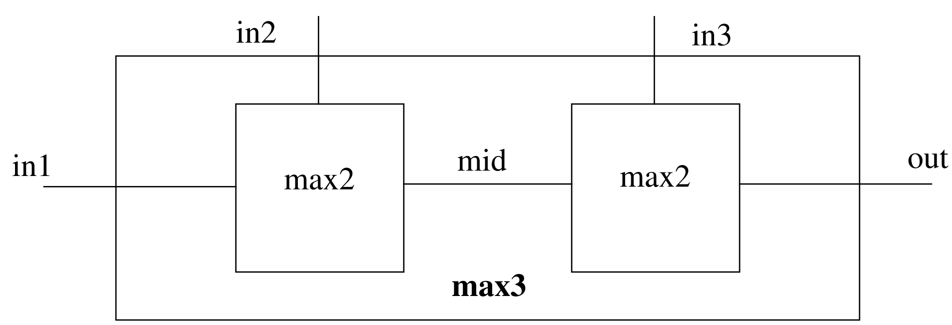
The main advantage of using an interpreter is to validate the actual behaviour of a specification with respect to its intended behaviour. Another important advantage involves test sequence generation.

At Ottawa University a LOTOS interpreter called **ISLA** (Interactive System for LOTOS Applications) has been implemented. This interpreter supports many functions to simulate the execution of a specification and to debug it. In the following we discuss only the main functions needed in our thesis. For more details about ISLA, references [Haj] and [GHL] can be consulted.

2.3.2 Step-by-Step Action

The basic operation of ISLA is to allow step-by-step execution of specifications. At each step, all possible actions are offered and the user chooses the action he wants to execute next and if this action involves a choice of data the user is asked to provide it. In this way one can check whether the actions conform to the ones expected by the designer of the specification.

The following example taken from [BB] illustrates the step-by-step action, by showing the execution of process max3. This process inputs three natural numbers in any order, through the three gates in1, in2 and in3, and outputs the maximum of these numbers, through gate out. Process max3 is composed by synchronizing over the hidden gate mid two instantiations of process max2. Process max2 inputs two natural numbers, in any order, through gates a and b, and outputs the maximum of these numbers through gate c. The following figure illustrates these processes.




```
process max3[in1,in2,in3,out] : noexit :=
  hide mid in
    (max2[in1,in2,mid] |[mid]| max2[mid,in3,out])
    where
      process max2[a,b,c] : noexit :=
        (a ? x : Nat; b ? y : Nat; c ! max(x,y);stop)
       []
        b ? x : Nat; a ? y : Nat; c ! max(x,y);stop
      endproc
endproc
```

First, the following three actions are offered by the interpreter. Note that the line numbers in the specification file of the offered actions are shown at the right of these actions.

```
==================================================================

 <1>- in1 ?x:Nat  ---> bh1  [17]
 <2>- in2 ?x:Nat  ---> bh2  [19]
 <3>- in3 ?x:Nat  ---> bh3  [19]

==================================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 1
 Enter a value for x:Nat => $2
==================================================================
```

Any one of the three natural numbers can be entered. The first action is chosen and the value 2 is entered through gate in1. The following two actions are offered:

```
==================================================================

 <1>- in2 ?y:Nat  ---> bh1  [17]
 <2>- in3 ?x:Nat  ---> bh2  [19]

==================================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 1
 Enter a value for y:Nat => $3
==================================================================
```

A natural number can be entered through either gate in2 or in3. The first action is chosen and the value 3 is entered through gate in2. The following two actions are then offered.

```
==================================================================
```

```
  <1>- in3 ?x:Nat  ---> bh1  [19]
  <2>- i (hiding: mid !$3:Nat)  ---> bh2  [17,17]

==================================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 2
   Internal event is executed
 Passed evaluated value ==> $3
==================================================================
```

In the first action offered, a natural number must be entered through gate in3. The second action is the result of the synchronization over gate mid of the two instantiations of process max2. This second action, in which no data is required to be entered, is chosen. Then, the following action is offered.

```
==================================================================

  <1>- in3 ?y:Nat  ---> bh1  [17]

==================================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 1
 Enter a value for y:Nat => $6
==================================================================
```

The value 6 is entered through gate in3 and the following action is offered

```
==================================================================

  <1>- out !$6:Nat  ---> bh1  [17]
```

==

```
ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 1
 Passed evaluated value ==> $6
```

==

The maximum value 6 which has been entered is output through gate out.

2.3.3 Process Tree

The step-by-step execution is inevitably slow, because of the interactions with the user. For this, it is possible to obtain the symbolic tree of a given process (or of the entire specification), where all execution paths are shown up to a maximum specified depth, by using symbolic execution. The predicates that involve variables whose values depend on interactions with the environment are assumed to be true and are shown in the tree.

Example

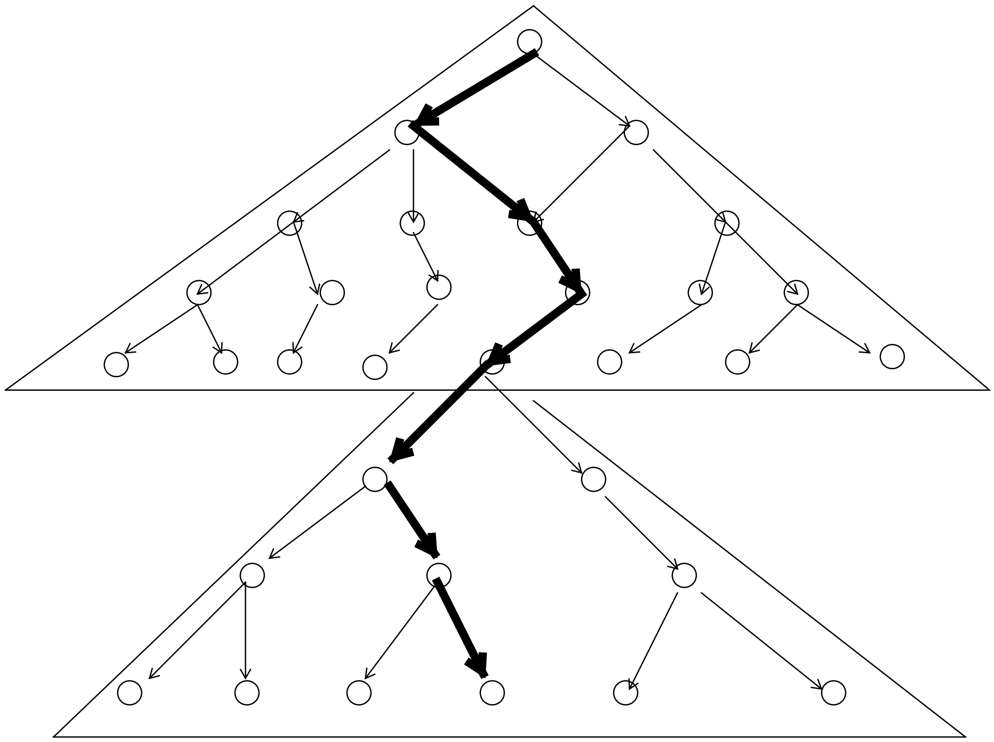
The symbolic tree of process max3 is given in the following. Note that this tree shows all possible orders of entering the three natural numbers. The numbers which appear in the right of each line of the tree are the line numbers in the specification file of the corresponding actions.

```
in1 ?x:Nat [17]
| 1 in2 ?y:Nat [17]
| | 1 in3 ?x:Nat [19]
| | | 1 i (hiding: mid !max(x,y):Nat) [17,19]
| | | | 1 out !max(x,max(x,y)):Nat [19] DEADLOCK
| | 2 i (hiding: mid !max(x,y):Nat) [17,17]
| | | 1 in3 ?y:Nat [17]
| | | | 1 out !max(max(x,y),y):Nat [17] DEADLOCK
| 2 in3 ?x:Nat [19]
| | 1 in2 ?y:Nat [17]
```

```
| | | 1 i (hiding: mid !max(x,y):Nat) [17,19]
| | | | 1 out !max(x,max(x,y)):Nat [19] DEADLOCK
2 in2 ?x:Nat [19]
| 1 in1 ?y:Nat [19]
| | 1 in3 ?x:Nat [19]
| | | 1 i (hiding: mid !max(x,y):Nat) [19,19]
| | | | 1 out !max(x,max(x,y)):Nat [19] DEADLOCK
| | 2 i (hiding: mid !max(x,y):Nat) [19,17]
| | | 1 in3 ?y:Nat [17]
| | | | 1 out !max(max(x,y),y):Nat [17] DEADLOCK
| 2 in3 ?x:Nat [19]
| | 1 in1 ?y:Nat [19]
| | | 1 i (hiding: mid !max(x,y):Nat) [19,19]
| | | | 1 out !max(x,max(x,y)):Nat [19] DEADLOCK
3 in3 ?x:Nat [19]
| 1 in1 ?x:Nat [17]
| | 1 in2 ?y:Nat [17]
| | | 1 i (hiding: mid !max(x,y):Nat) [17,19]
| | | | 1 out !max(x,max(x,y)):Nat [19] DEADLOCK
| 2 in2 ?x:Nat [19]
| | 1 in1 ?y:Nat [19]
| | | 1 i (hiding: mid !max(x,y):Nat) [19,19]
| | | | 1 out !max(x,max(x,y)):Nat [19] DEADLOCK
```

2.3.4 Behaviour Tree

Since large trees are impractical to compute and read, and since they contain many uninteresting paths, a tree is computed to a certain depth and the one-stepper method is then used to reach a specific leaf in the tree. A tree is then computed from that node, and the same process is repeated [Haj]. The figure given in the following shows this method. This technique was used in the test sequence derivation of our LAPB LOTOS specification.



2.4 Specification Styles in LOTOS

2.4.1 Introduction

The design of a specification requires to find an appropriate structure for its architecture; for complex systems this task is quite difficult. Different specification styles that allow to structure formal specifications can be used. In the following, we discuss briefly the use of different styles for designing LOTOS specifications. A more detailed discussions, about specification styles can be found in [VSS].

2.4.2 The Constraint Oriented Style

The most popular one, the constraint-oriented style, allows to write implementation-independent specifications by separating concerns. This style, in which the focus on the observable ordering of events is defined by a conjunction of separate constraints, permits to produce modular, well structured specifications.

However, because constraint decomposition is done in such a way as to maximize logical clarity rather than showing an implementation structure, it may be difficult to understand the overall behaviour of the specification from the knowledge of its components and relationship between them. The abstractness of this style makes specifications hard to implement. The simulation task of such specifications can be tedious, mainly because of the use of the *choice* construct and the high level of parallelism (see 2.5.2). Also, since the separation of concerns stems largely from the use of parallel operators, the symbolic tree of such specifications will be very large (especially for large specifications) giving all the combinations of predicates which correspond to constraints and many of these predicate combinations may be contradictory. This last point is illustrated by the following example [LG]:

```
( in ? x:Nat [x > 3]; out ? y:Nat [y > x]; exit
 []
 in ? x:Nat [x <= 3]; out !3 ;exit  )
||
 ( in ? x:Nat; (out ? y:Nat [y < 3]; exit
              []
              out ? y:Nat [y = x]; exit) )
```

```
1 in ?[x,x]:Nat [x > 3]
| 1 out ?[y,y]:Nat [y > x] [y < 3]
| | 1 exit ** EXIT SUCCEED **
| 2 out ?[y,y]:Nat [y > x] [y = x]
| | 1 exit ** EXIT SUCCEED **
2 in ?[x,x]:Nat [x < 3]
| 1 out !3:Nat [3 < 3] **false**
| 2 out !3:Nat [3 = x]
| | 1 exit ** EXIT SUCCEED **
```

In the symbolic tree of this example, it can be seen that branches (1.1) and (1.2) contain predicates which are always false (because of contradictory conditions). Since the interpreter is not able to evaluate predicates which contain variables, these branches are shown in the tree. The uniform detection of contradictions is actually an undecidable problem (see section 4.5.2.3). Note that branch (2.1) can be pruned by the interpreter because its predicate contains no variables and can be evaluated to false.

2.4.3 The Monolithic Style

In this style, all possible sequences of actions are shown explicitly without distinguishing parts for structuring the specification. The specification appears like a tree of alternatives.

This style can be very useful for simple specifications but it is of little practical use for designing complex specifications because of its lack of structure, its difficulty for human comprehension and for the fact that it results in specifications of large size. Nevertheless, it can be very useful for the derivation of test cases.

2.4.4 The State Oriented Style

In this style, the system is represented by a single resource whose internal state space is explicitly defined.

Like the monolithic style, this style does not result in well structured specifications. Therefore, it is not suitable for specifications where abstractness is needed. However, it is particularly useful for specifications which are wanted to be relatively close to an implementation. But, for complex systems the identification of a multiplicity of state variables describing the state space can be a quite difficult task. Informal or semi-formal specifications often include state tables; the state-oriented style can be useful to translate these tables faithfully into LOTOS.

2.4.5 The Resource Oriented Style

In this style, the system is described by different resources which interact among themselves through interfaces. Each resource can be specified using a constraint oriented, monolithic or state oriented approach allowing to iteratively apply the different styles.

This style allows modularity and parallel structures and usually specification modules correspond to implementation modules.

2.4.6 Mixing Different Styles

From the previous discussions, it can be said that different styles meet different needs. Consequently, depending on the goal of the specification designer, the specification may use the constraint oriented, the monolithic, the state oriented, the resource oriented or a mixture of these styles. However, it should be noted that the four described styles are actually extremes in the sense that specifications of 'real' systems are normally based on a mixture of these styles [VSS].

2.5 Some LOTOS Constructs

2.5.1 Introduction

In the following we present some LOTOS constructs. Some of these constructs are frequently used in existing LOTOS specifications. Others are derived from our LAPB specification and are, from our point of view, quite interesting to present.

A shortcoming of LOTOS, as it is defined now, is that it is not possible to write specifications using other predefined specifications. This feature could be easily added to LOTOS and may be introduced in the near future. The point is to look for frequently used specification parts and define them as built-in processes in a library. Some of the constructs described in the following may be used for this purpose. We shall, also, use simplified LOTOS syntax (e.g. we shall omit gate parameters) to improve readability.

2.5.2 Dischoice Construct

In the constraint-oriented specification style, one often finds constructs such as the following:

process releasequeue1 (nr : Nat, q : pqueue) : **exit**(pqueue) :=

    [not(isempty(q))] ->

```
        (choice newrq : pqueue, f : eframe []
          [q eq (newrq --+ f)] ->
            ([snn(f) ne nr] -> releasequeue1(nr, q)
             []
             [snn(f) eq nr] -> exit(newrq)
            )
        )
      []
      [isempty(q)] -> exit(q)
endproc
```

Given a queue *q* of elements of type *eframe* and a natural number *nr*, the process *releasequeue1* removes from the queue *q* all the elements *f* such that the field *snn(f)* is different from *nr*, until an element *f* for which field *snn(f)* is equal to *nr* is found.

This construct may seem 'elegant' but it is quite tedious to simulate with an interpreter. This is illustrated by the following execution example of process releasequeue1. To simplify we assume that the type eframe represents octets. Each element of the queue q consists of an octet and a natural number. The operation snn(f) gets the natural number of a queue element. We assume, also, that the first time process releasequeue1 is instantiated, it is given as parameters nr equal to 3 and q as shown below:

3	11101001	2	10101010	1	11110000

The first two elements with the *snn(f)* field respectively equal to 1 and 2 must be removed. The execution with an interpreter gives first the following sequence of actions:

```
=================================================================

* <1>- value for newrq:pqueue is needed for 'choice'   ---> bh1  [64]

=================================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
```

==> 1

For "choice", value for newrq:pqueue must be entered
==> --+(--+(empty,m($3,Octet(1,1,1,0,1,0,0,1))),m($2,Octet(1,0,1,0,1,0,1,0)))
==
As shown above the data newrq is entered as equal to: | 3 | 11101001 | 2 | 10101010

==

* <1>- value for f:eframe is needed for ’choice’ ---> bh1 [64]

==

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 1

For "choice", value for f:eframe must be entered
==> m($1,Octet(1,1,1,1,0,0,0,0))
===

The data f is entered as equal to ($1,Octet(1,1,1,1,0,0,0,0)). The field snn(f) (equal to 1) is not equal to 3, therefore process releasequeue1 calls itself with parameters 4 and newrq as shown before. In the next two actions (which are the same as the previous two ones):

-- the data f is entered as equal to: | 2 | 101010101 |

--the data newrq is entered as equal to: | 3 | 11101001 |

So for the actions offered which require data to be entered, we have to figure out what queue newrq and what element f to provide such that q is equal to (newrq --+ f). This can be a difficult task if the number of elements of q is important. If executability of specifications is desired, the following construct can be used instead:

process releasequeue1 (nr : Nat, q : pqueue) : **exit**(pqueue) :=

```
        [isempty(q)]  -> exit(q)
        []
        [not(isempty(q))] ->
            ([(nr ne (snn(firstone(q))))] ->
                    i
                    ; releasequeue1 (nr, removefirst(q))
            )
            []
            [nr eq (snn(firstone(q)))] ->
                        exit(q)
endproc
```

This specification is equivalent to the previous one but is very easy to simulate because it is really a recursive program. No need of user interactions exists. Although the 'choice' construct may result in shorter specifications, because of the fact that a simulation was extensively used in this work, this construct was avoided. The operations *firstone* and *removefirst* which respectively get the first element of *q* and remove the first element of *q* are defined in the sort declaration of *pqueue*. Note the use of the explicit internal event *i* before process releasequeue1 calls itself recursively, in order to avoid infinite looping.

2.5.3 Exit and Disable Constructs

When several processes synchronize among them and one process dictates the exit condition to the others, the following construct where P dictates the exit to Q can be used:

```
P := [condition c1] -> g ! x; P                    Q := g ? x; Q
        []                                               [>
        [condition c2] -> exit                          exit
```

In P||Q, when P decides to exit, it will force a rendez-vous with the exit of Q. In other words, process Q will have to execute the disable and exit. This construct was not used in our case. Another similar construct is used when several processes synchronize between them, each one checking a different constraint, and one of them dictates the exit condition. For example:

```
P := g ? x [x gt 5]; P
   []
     g ! 0; exit
```

```
Q := g ? x [(x mod 3) eq 0]; Q
     [>
       exit
```

Process P||Q accepts only numbers greater than 5 and multiples of 3, and when 0 is encountered the exit of P will force a rendez-vous with Q. Although this construct is correct, each instantiation of Q creates a new disable so that P synchronizes with the action *g ? x [(x mod 3) eq 0]* disabled by a certain number of nested *exit* of Q; the choice of any one of the offered exits produces the same result, i.e exiting of both processes. This is illustrated by the following execution of P||Q. First, the interpreter offers the following two actions:

```
================================================================

 <1>- g ?[x,x]Nat
      [ge(x,$5)]
      [eq(x mod $3,0)]   ---> bh1  [22,27]
 <2>- g !0:Nat
      [eq(0 mod $3,0)]   ---> bh2  [24,27]

================================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 1
  The existing predicate(s) for this action :
  [ge(x,$5)]
  [eq(x mod $3,0)]
 Enter a set or a value for [x,x]Nat => $9
================================================================
```

The first action is chosen and the value 9 is provided for the variable x. Note that P and Q call themselves after executing the first action, resulting in the second instantiation of these two processes. The previous two actions are offered again, giving:

```
=================================================================

  <1>- g ?[x,x]Nat
      [ge(x,$5)]
      [eq(x mod $3,0)]   ---> bh1  [22,27]
  <2>- g !0:Nat
      [eq(0 mod $3,0)]   ---> bh2  [24,27]

================================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 2
   The existing predicate(s) for this action :
   [eq(0 mod $3,0)]  evaluated to true
 Passed evaluated value ==> 0
=================================================================
```

The second action g ! 0:Nat is chosen. After this action is executed, process Q calls itself, resulting in its third instantiation. Now the following three actions are offered:

```
=================================================================

  <1>- exit  ---> bh1  [24,30]
  <2>- exit  ---> bh2  [24,30]
  <3>- exit  ---> bh3  [24,30]

=================================================================
```

From this we can see that three synchronized exit actions are offered. These are the result of the synchronization of the exit action of P with the disable exit of each one of the three instantiations of Q. Any one of these exit actions can be chosen, yielding exiting from the processes. This construct was not used in our LAPB specification.

However, one must be careful when using the previous construct in cases where P and Q have parameters and exit values. Assume that P exits the queue *queuex* of integers x such that x greater than 3 and Q outputs the queue *queuey* of integers y such that y greater than 1 but as soon as x equal to 0 is encountered P and Q must exit *queuex* and *queuey*; this could be written as shown below :

```
P(queuex) := g ? x ? y [x gt 3];P(x +-- queuex)
             []
              g ! 0 ? y; exit(queuex, any queue)

Q(queuey) := g ? x ? y [y gt 1];Q(y +-- queuey)
             [>
              exit(any queue, queuey)
```

As in the previous case, the action *exit(queuex, any queue)* of P synchronizes with each one of the nested *exit(any queue, any queue)* and now the choice of an action between the offered actions of synchronized exits is not the same since for each action, *queuey* has a different value. This is illustrated by the following execution example of process P||Q. First, the following two actions are offered:

```
==================================================================

 <1>- g ?[x,x]Nat ?[y,y]Nat
        [gt(x,$3)
        [gt(y,$1)]   ---> bh1  [42,51]
 <2>- g !0:Nat ?[y,y]Nat
        [gt(y,$1)]   ---> bh2  [46,51]

==================================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 1
  The existing predicate(s) for this action :
  [gt(x,$3)]
  [gt(y,$1)]
```

```
 Enter a set or a value for [x,x]Nat => $4
 Enter a value for [y,y]Nat => $2
===============================================================
```

The first action is chosen and the values 4 and 2 are entered for respectively x and y. After executing this action, P and Q call themselves resulting in their second instantiation and the same two actions are offered again:

```
===============================================================

  <1>- g ?[x,x]Nat ?[y,y]Nat
          [gt(x,$3)]
          [gt(y,$1)]   ---> bh1  [42,51]
  <2>- g !0:Nat ?[y,y]Nat
          [gt(y,$1)]   ---> bh2  [46,51]

===============================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 2
   The existing predicate(s) for this action :
   [gt(y,$1)]
 Passed evaluated value ==> 0
===============================================================
```

The second action g !0:Nat is chosen. After this action is executed, process Q calls itself resulting in its third instantiation and the following three actions are offered:

```
===============================================================

  <1>- exit !--+(empty,$4):nqueue !--+(--+(empty,$3),$2):nqueue  ---> bh1
 [47,54]
  <2>- exit !--+(empty,$4):nqueue !--+(empty,$2):nqueue  ---> bh2  [47,54]
  <3>- exit !--+(empty,$4):nqueue !empty:nqueue  ---> bh3  [47,54]

===============================================================
```

From this we can see that three synchronized exit actions are offered. In each action the variable queuey has a different value. This is the result of the synchronization of the exit of P with the exit of each of the three instantiations of Q. For each instantiation, queuey has a different value. Therefore, we have a situation of nondeterminism where different exits cause different values to be passed to the enabled process. This construct was not used in our specification.

Another exit construct is when any one of the processes is able to terminate all others. Suppose that any of the three processes P, Q and R must be able to terminate all others; this could be written as:

```
P := ( ... ;P) [] [condition] -> exit(1)
    [> (exit(2) [] exit(3))

Q := ( ... ;Q) [] [condition] -> exit(2)
    [> (exit(1) [] exit(3))

R := ( ... ;R) [] [condition] -> exit(3)
    [> (exit(1) [] exit(2))
```

Unfortunately each process must know the exit parameters of all others which is not really modular. Also if we have n processes, (n-1) exits must appear in the disable part of each process; this is quite inconvenient if n is large. Note that putting the *exit(any)* action in the disable part of each process would not be correct, because then it would be possible for all processes to synchronize on exit at the very beginning. In this case, processes P, Q and R would be written as process P below.

```
P := ( ... ;P)
      [> exit(any)
```

Another way to resolve this problem is to introduce a new gate p and assign to each process a different number; this could be written as:

```
P := ( ... ;P) [] [condition p1] -> p ! 1; exit
```

```
        [> p ? x [(x ne 1) and (x le n)]; exit

Q := ( ... ;Q) [] [condition q1] -> p ! 2; exit
        [> p ? x [(x ne 2) and (x le n)]; exit

R := ( ... ;R) [] [condition r1] -> p ! 3; exit
        [> p ? x [(x ne 3) and (x le n)]; exit
```

Some people could say that this solution is not as elegant as the previous one; however it provides modularity and a shorter code. The modularity is due to the fact that a process does not have to know the number assigned to the other processes. This construct was not used in our LAPB specification.

The following construct has been used in the LAPB specification. A process S contains three processes P, Q and R. Process P can, in two cases, terminate its execution and activate a process Q. The first case is when Q disable P. The second case is when P synchronizes, through a hidden gate p, with process R, which in its turn, synchronizes with Q. This has been written as:

```
S := (P(0,n2)
       [>
         Q)
      |[p]|
         R

P(n1,n2) := [n1 lt n2] ->  ( ... ;P(n1 +1,n2))
                    []
                    [n1 eq n2] -> p ! lreset; stop

Q := phl ? f [condition q1] ( ... )
        []
        p ! causelreset; ( ... )

R := p ! lreset
```

;p ! causelreset

;exit

Process P has as parameters n1 and n2. The first time it is instantiated, the value of n1 is 0 and each time P calls itself n1 is incremented. Process P terminates its execution and process Q starts its execution in two situations. The first one is, by way of disable, when Q receives, through gate phl, a data f which fulfills condition q1. The second situation is when P has called itself n2 times. In this case, P synchronizes with R over the action *p ! lreset*. R, in its turn, synchronizes with Q over the action *p ! causelreset*, enabling Q to start execution. In the LAPB specification, process S corresponds to process *dataphase,* process P corresponds to process *inforphase*, process Q corresponds to process *linkreset* and process R corresponds to process *causelinkreset.*

Chapter 3. Design of the LAPB Specification in LOTOS

3.1 Introduction

This section describes our LOTOS specification of the X.25 data link layer LAPB defined in the ISO/DIS 7776 [ISO8]. The ISO/DIS 7776 specifies the X.25 level two 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.

LAPB specification 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 in 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 (see 4.5.4), we made it visible. Figure 3.1 shows the gate configuration of the specification.

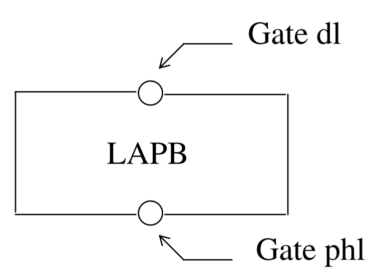


Figure 3.1- Gate Configuration of the Specification

3.2 LAPB Architecture

The generation of test sequences for the LAPB specification is based on the use of the symbolic tree. The specification is quite large, so for our purposes it is not practical to write the whole specification as separate constraints because this will result in huge symbolic trees. For this reason the specification is separated in different phases (Connection Phase, Data Phase and Termination Phase) of a LAPB DTE; then for each phase symbolic trees are generated.

For the Connection Phase, *waitingresponse* and *waitresponsedm* processes are written as several processes participating in a multiway rendez-vous because the obtained tree is manageable since these processes are not too large.

Beside the fact that we wanted the symbolic trees to be of reasonable size, we wanted also the specification to be relatively close to an implementation. For this reason, state variables have been considered in the Data Phase (process *infphase*).

A point worth mentioning is the treatment of timers. A timer is described as a process, as shown below, which is composed in parallel with the processes that use it, and share with it 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 adequate for our problem.

```
process Timer[t] : exit :=
  t ! start
  ;(i
    ; t ! expired; exit
   [> t ! reset; exit)
endproc
```

3.3 Validation of the LAPB Specification.

This specification has been extensively tested using the LOTOS interpreter of the University of Ottawa and as far as we were able to determine is consistent with the ISO/DIS 7776 LAPB description.

First, the three main components of the specification, the link set-up phase, the data transfer phase and the termination phase, were tested separately, by doing step-by-step action. In each phase, we tested all the proper Protocol Data Units (PDUs) (i.e. PDUs which are syntactically correct and also acceptable at the current state of the DTE), since the number of such PDUs is limited. In each phase, also, we tested some inopportune PDUs (syntactically correct PDUs but arriving at an inappropriate state of the DTE), since the number of such PDUs is quite important compared with the number of proper PDUs. Concerning improper PDUs (syntactically incorrect PDUs), we tested one PDU per condition in each phase. For example, for the condition *incorrect PDU length*, we tested only an RR frame with an incorrect length.

The link set-up phase and the termination phase are quite simple. The data transfer phase is more complex. For this reason, we again divided this phase into two parts: the normal data transfer phase and the link reset phase, and we tested each part separately.

After the different parts of the specification were tested, tests were done on the whole specification. For this, some proper PDUs were used.

One can note, also, that the generated symbolic execution trees, used in test case generation, were another tool for specification validation. These trees were thoroughly compared with the LAPB state tables of [KLPU].

3.4 LAPB Specification

The data link layer provides the DTE with three basic functions:

-- link initialization: necessary for the DTE to begin communication in a known state.

-- flow control: controls the flow of frames between the DTE and the other station.

-- error control: provided in two forms:

-- a cyclic redundancy check.

-- use of sequence numbers to ensure against losing entire frames.

The DTE communicates with the remote entity by sending and receiving frames; there are nine different frames separated in three groups. This is shown in Table 3.1.

Type	Symbol	Meaning
I	I	Information (contains the data)
S	RR	Receive Ready
	RNR	Receive Not Ready
	REJ	REJect
U	SABM	Set Asynchronous Balance Mode
	SABME	Set Asynchronous Balance Mode Extended
	DISC	DISConnected
	UA	Unnumbered Acknowledgement
	DM	Disconnect Mode
	FRMR	FRaMe Reject

Table 3.1 - Frame Types

The formal description of LAPB is a parameterized specification with five parameters: *n2, k* , *resolvecollis*, *senddmoption* and *tmode*. The value of *n2* indicates the maximum number of attempts that are made by the DTE to complete the successful transmission of a frame to the remote entity. The value of *k* indicates the maximum number of sequentially numbered I frames that the remote entity may have outstanding (i.e unackowledged) at any given time. The value of *k* does not exceed seven for modulo eight operations and one hundred and twenty seven for modulo one hundred and twenty eight operations. The variable *resolvecollis* indicates what kind of actions to be done by the DTE in case of collision of frames during Link Set Up Phase or Disconnection Phase (see process *collision*). The boolean parameter *senddmoption* is true in case where the DTE has the option of asking the remote entity to set up the link. The parameter *tmode* indicates which one of the two operation mode is supported by the DTE:

--basic operation (modulo eight).

--extended operation (modulo one hundred and twenty eight).

The specification is structured as shown in the following:

```
specification     X.25linklayer[dl,phl]    (n2,k,resolvecollis : Nat, senddmoption : Bool,
                                                                        tmode : mmode) : noexit
    library
         NaturalNumber, Boolean, Bit, BitString
    endlib

       < data part description >

   behaviour

       < behaviour part description >

endspec
```

3.4.1 Data Type Part

For each type, the starting and ending line numbers of the type as shown in the Appendix A, are given between brackets.

Primitive Type (427-470)

The communication between LAPB and X.25 is done by means of these primitives. The auxiliary function *eva*l that maps a primitive to a natural number is defined in order to simplify the specification of the boolean operators defined on the primitives. The primitives *dl_data_req* and *dl_data_ind* have as parameter a packet. A packet is defined as a string of bits.

Frame Type (205-335)

The figure below shows the structure of an I frame:

flag	address	control	packet	frame check sequence (FCS)	flag

The figure below shows the structure of an S or U frame:

flag	address	control	frame check sequence (FCS)	flag

The auxiliary function h that maps a frame to a natural number is defined in order to simplify the specification of the boolean operators defined on the frames. The different functions make allow to build a supervisory, an unnumbered or an information frame, given the components of the frame. The boolean functions i*s* determine the type of a given frame.

Flag Type (95-106)

All frames must start and end with the flag field which is a constant known value. The operation *flagging* represents a flag. The function *noflagging* makes a flag to be incorrect. The boolean function *is_flag* determines if the flag is correct or not.

Address Type (69-93)

The address field identifies a frame as either a command or a response frame. A command frame contains the address of the station to which the command is being sent. A response frame contains the address of the station sending the frame.

Command frames sent by the DTE contains the address "b" while response frames sent by the DTE contains the address "a". The functions *is_a* and *is_b* determine the content of the address field of a frame and therefore if it is a command frame or response frame. The operation *corrupt_adr* allows to make an address corrupted and the boolean function *is_corrupt_adr* determines if an address is corrupted or not.

Control Type (157-183)

The control field indicates the type of commands or responses and contains sequence number where applicable. Three types of control field corresponding to the three types of frames, I, S and U frames, are considered. The structure of the control field is shown in figure 3.2. The three functions *ctl_iframe*, *ctl_sframe* and *ctl_uframe* get the control field of respectively an I frame, an S frame and a U frame. Ctl_iframe has as parameters NS, P and NR, ctl_sframe has as parameters P and NR and ctl_uframe has as parameters P/F.

These parameters have the following meaning:

-- NS: transmitter send sequence number.

-- NR: transmitter receive sequence number.

-- P/F: poll bit when issued as a command and final bit when issued as a response.

-- P: poll bit.

-- M: modifier function bit.

-- S: supervisory function bit.

The function *is_undef_ct*l determines if a control field is undefined or not. The three functions *nss, nrr* and *pff* get respectively NS, NR and PF fields of a given frame.

<table>
<tr><td>Control Field Bits</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>I format</td><td>0</td><td colspan="3">NS</td><td>P</td><td colspan="3">NR</td></tr>
<tr><td>S format</td><td>1</td><td>0</td><td>S</td><td>S</td><td>P/F</td><td colspan="3">NR</td></tr>
<tr><td>U format</td><td>1</td><td>1</td><td>M</td><td>M</td><td>P/F</td><td>M</td><td>M</td><td>M</td></tr>
</table>

Figure 3.2 - Control Field Structure

Frame Check Sequence Type (FCS) (185-196)

FCS is used for error control. The operation *fc* represents a non corrupted FCS; the function *corrupt_FCS* makes a FCS corrupted and the boolean function *is_corrupt_fcs* determines if a FCS is corrupted or not.

Frmrreason Type (198-203)

This field is used by the DTE in an FRMR response frame to report an error not recoverable by retransmission of the identical frame. The structure of this field is shown in figure 3.3. The fields Nr, NS, C/R, W, X, Y and Z are indicators about the frame which causes the rejection condition.

Bits: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Control Field	0	NS	C/R	NR	W	X	Y	Z	0	0	0	0

Figure 3.3 - Frmrreason Field

Extended frame Type (337-425)

The *Extframe* type is an extension of the type *Frame*. The *m* operator allows to make an extended frame by adding to the frame four fields *inforr, corrlength, lesslength* and *abortf* which specify respectively if the frame contains information field, if the frame length is correct, if the frame length is not less than the minimum frame length and if the frame is not aborted.

The operators *mod* and - define respectively the modulo and subtraction operations on natural numbers. The operations *flg1, flg2, pack, adrs, ctrl, fcss, reasfrmr, ns, nr*, and *pf* permit to get respectively the first flag, the second flag, the packet, the address, the control, the frame check sequence, the frmr reason, the ns, the nr and the pf bit fields of a given eframe. The operations *get_f, get_c, get_l* and *get_a* allow to get respectively the frame part, corrlength, lesslength and abortf fields of a given eframe.

The predicate *is_valid* determines if a frame is valid or not. An invalid frame is defined as one which:

-- is not properly bounded by two flags.

-- or the frame length is less than the minimum frame length.

-- or the FCS is incorrect.

-- or the address field is corrupted.

The predicate *is_reject* determines if a frame is to be rejected or not. This predicate is true if at least one of the following conditions occurs:

-- receipt of an undefined control field in a given frame.

-- receipt of an information field with incorrect length.

-- receipt of frame with an invalid NR field.

-- receipt of a frame with an information field which is not permitted.

The predicate *not_valid_nrf* determines if the NR field of a given frame is valid or not. An invalid NR is defined as one which points to an I frame which has been transmitted and acknowledged or to an I frame which has not been transmitted and is not the next sequential I frame pending transmission.

The predicates *not_abort* and *corrl* determine respectively if a frame has been aborted and if the frame length is correct. The predicate *correct* determines if a frame is correct or not. A frame is correct if it is valid, it is not to be rejected and is not aborted. The predicate *eq* tests if two frames are equal. The predicate collis is true if two frames have the same type. The predicate *resp(f)* is true if the frame f is a DISC frame and is false if f is SABM or SABME frame.

Extended packet Type (510-522)

The type *Packe* defines an extended packet type. It allows to assign to each packet a natural number. The *mp* operation permits to make such an extended packet while *snn* and *packe* operations get respectively the natural number and the packet fields of the extended packet.

Extended packet Queue Type (530-536)

The type *Pqueue* defines a queue of extended packets.

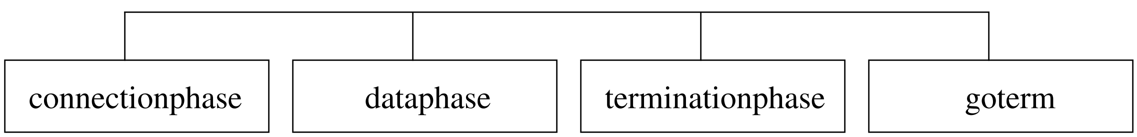
3.4.2 Behaviour Part

General Form of the Specification

The general form of the specification is given by *LAPB* process (line 557 to line 2367) which distinguishes between the different phases of the DTE communication.

If the Connection Phase is successful done, as reported by the boolean variable *ok*, the protocol enters the Data Phase. The *dataphase* process can be disabled by the *terminationphase* process. The process *dataphase* can stops itself by synchronizing with process *goterm* which in its turn synchronizes with process *terminationphase*. The boolean variable *donotwaitua* which specifies the type of connection call used by the DTE, is passed as a parameter to *dataphase* process, *from* and *to* parameters specify addresses.

The following figure shows the different processes which constitute process *LAPB*:



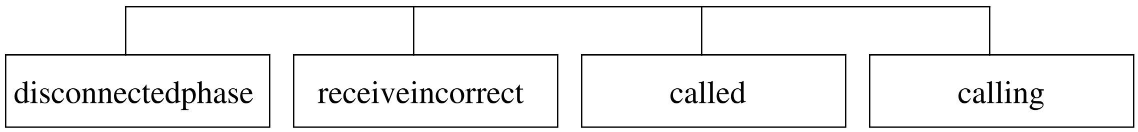
For each process the start and end line numbers where this process appears in the specification are given between brackets.

Connectionphase Process (580-834)

This process represents the DTE at the Disconnected State; it is composed of four alternatives:

- -- *disconnectedphase* process: describes the case where the DTE receives frames which are canceled at the Disconnected State; the DTE remains at the Disconnected State.
- -- *receiveincorrect* process: describes the case where the DTE receives an incorrect frame; the DTE ignores this frame and remains in the Disconnected State.
- -- *called* process: describes the case where a SABM or a SABME frame (establishment request) or a DM frame with final bit equal to 0 is received from the remote entity.
- -- *calling* process: describes the case where the DTE receives from the upper layer an establishment request primitive.

The figure below shows the processes constituting process *connectionphase*:



Disconnectedphase Process (618-636)

Received frames which are not SABM, SABME or DM with final bit equal to 0, are canceled. For those command frames with poll bit set a DM frame with final bit set is sent. If a DISC frame is received a response DM frame is sent.

Called process (594-616)

If the DTE receives a SABM or a SABME frame as an establishment request, it can either accept it by sending a UA frame or refuse it by sending a DM frame.

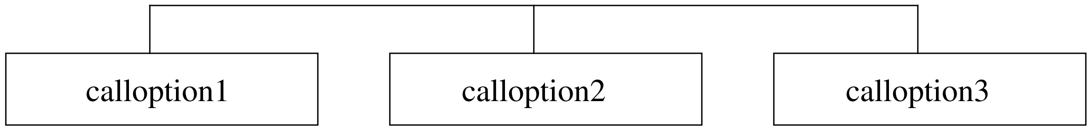
If the DTE receives a DM frame with final bit equal to 0 as a request for initiating Link Set Up, it can either accept it by sending a SABM or a SABME frame (this is done by *calloption1* process), or refuse this request by sending a DISC frame.

Calling Process (637-651)

The DTE can initiate Link Set Up in three manners:

- -- send a SABM or a SABME frame; this is done by calling *calloption1* process and giving it SABM or SABME as a parameter.
- -- initiate link disconnection by sending a DISC frame, then initiate Link Set Up by sending a SABM or a SABME frame; this is done by calling *calloption2* process.
- -- request the remote entity to initiate Link Set Up by sending an unsolicited DM response; this is done by *calloption3* process.

The figure below shows the processes composing process *calling*:



Calloption1 Process (1133-1170)

The given parameter frame f (SABM or SABME) is sent, then a response to f is waited. This is done by synchronizing over the gate phl and the hidden gates pp and p process *sendcde* which sends f and process *waitresp* which waits for the response to f.

Sendcde process (1093-1131)

A frame f is sent over gate phl, the timer is started and process *synchresp* is called to synchronize with process *waitresp*. A frame f may be sent up to *n2* times before a response frame is received.

In case where frame f has been sent *n2* times without receiving a response frame action *exit(false, false)* is done to report unsuccessful transmission of f. Parameter *init* indicates the number of times f was sent without receiving a response.

Synchresp Process (1112-1130)

This process is composed of three alternatives:

-- the first alternative corresponds to the case where the received frame is ignored.

-- the second alternative is the case where the received frame is not a SABM or a DISC frame; so the timer is reset and an exit is executed.

-- the third alternative is the case where a SABM, a SABME or a DISC frame is received after having sent frame f; the timer is reset, a frame is sent and then either an exit is executed in the case where the received frame does not result in a collision with frame f (i.e received frame is not the same as sent frame f), or process *wwaitua* is called in case of collision.

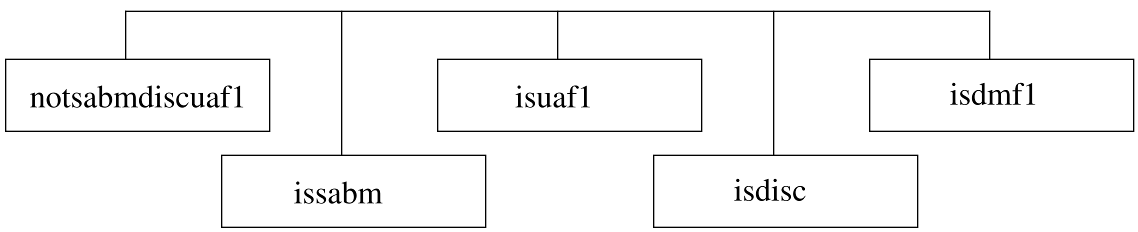
Waitresp Process (1148-1169)

After the first two actions which correspond to the sending of frame f by *sendcde* process are done, the process *waitingresponse* is called. The disable part corresponds to the case where frame f has been sent *n2* times without receiving a response.

Waitingresponse Process (1156-1168)

Processes *notsabmdiscuaf1dmf1, issabm, isdisc, isuaf1* and *isdmf1* participate in the reception of the response to the sent frame f, by synchronizing over gates phl, pp and p.

The following figure shows processes composing process *waitingresponse*:



Notsabmdiscuaf1 Process (881-910)

The first alternative corresponds to the case where the received frame is not correct or is not a SABM frame and is not a SABME frame and is not a DISC frame and is not a UA frame and is not a DM frame with final bit equal to 1; in this case the process ignores the received frame by calling itself.

The second alternative corresponds to the case where the received frame is correct and is a SABM frame, a SABME frame, a DISC frame, a UA frame or a DM frame with final bit equal to 1. The first disable occurs in case of reception of a SABM frame, a SABME frame or a DISC frame.

The second disable corresponds to the case where the time expires and another frame f must be sent; so process *notsabmdiscuaf1dmf1* calls itself.

Isuaf1 Process (912-935)

The first alternative corresponds to the case where the received frame is a UA with final bit equal to 1. The second alternative is the case where the received frame is not correct or is not a UA with final bit equal to 1. The first disable corresponds to the case where the received frame is a SABM, SABME or DISC frame. The second disable corresponds to the case where the time expires and another frame f must be sent.

Isdmf1 Process (937-960)

This process has the same structure as process *isuaf1* except that the received frame is a DM with final bit equal to 1. Note that the boolean value of *resp(fr)* in the action *exit(resp(fr), false)* is false if the sent frame fr is SABM or SABME and true if fr is a DISC frame.

Issabm Process (962-996)

The first alternative is the case where the received frame is not correct, is not SABM or is not SABME frame. In the second case the received frame is SABM or SABME; if there is no collision between the sent frame fr and the received SABM or SABME (i.e the sent frame is a DISC), the actions following the guard *[not(coll)]* are done in order to send a DM frame. If a collision results, process *collision* is called. The first disable corresponds to the case where the received frame is a DISC. The second disable is the case where the time expires and another frame fr must be sent.

Isdisc Process (997-1031)

This process has the same structure as process *issabm* except that the received frame is a DISC.

Collision Process (1033-1047)

A UA frame is sent, then depending on the type of collision indicated by the variable *resolvecollis*, the DTE will enter the indicated phase (Data Phase or Disconnected Phase) after one of the following events

-- after receiving a UA response; this is done by process *waitua*.

-- after timing out; this is done by process *waituaortimeou*

-- directly after having sent the UA frame; this is done by process *donotwait*.

Calloption2 Process (652-669)

This process first calls *calloption1* process giving it a DISC frame as parameter to disconnect the link then calls again *calloption1* process giving it a SABM or a SABME frame as parameter to initiate Link Set Up.

Calloption3 Process (671-737)

In this case a DM frame with final bit equal to 0 is sent by the DTE as a request to initiate Link Set Up. This process is composed by the synchronization over the gate phl and the hidden gates pp and p of processes *senddm* and *waitrespdm*.

Senddm Process (679-716)

A DM frame is sent, the timer is started and process *synchrespdm* is called to synchronize with process *waitrespdm*. The DM frame may be sent up to *n2* times if the time expires without having received a response frame. The action *exit(false, false)* is done to report unsuccessful transmission of a DM frame in case of *n2* times transmission of a DM frame without reception of a response frame. Parameter *init* indicates the number of times a DM frame was sent before receiving a response.

Synchrespdm Process (699-715)

This process is composed of three alternatives. In the first case the received frame is ignored. The second alternative is the case where the received frame is a DISC, a SABM or a DM with final bit equal to 1; so the timer is reset and an exit is executed. The third alternative is the case of collision (i.e the received frame is a DM with final bit equal to 0); so either an exit is executed or process *wwaitua* is called.

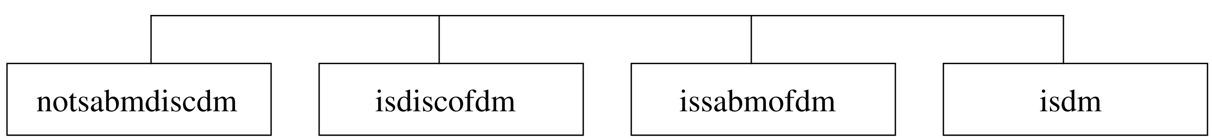
Waitrespdm Process (717-736)

This process waits first for the sending of frame f then process *waitingresponsedm* is called. The disable part corresponds to the case where the frame f has been sent *n2* times without receiving a response.

Waitingresponsedm Process (725-735)

Processes *notsabmdiscdm*, *isdm*, *issabmofdm* and *isdiscofdm* participate in the reception of the response to the sent DM frame, by synchronizing over gates phl, p and pp.

The figure below shows the processes composing process *waitingresponsedm*:



Notsabmdiscdm Process (739-755)

The first alternative is the case where the received frame is not a SABM, a SABME, a DISC or a DM; so it is ignored. The second alternative is the case where the received frame is a SABM, a SABME, a DISC or a DM frame. The disable part is the case where the time expires and so another DM frame must be sent.

Isdiscofdm Process (757-774)

The first alternative is the case where the received frame is a DISC frame; and *exit(false, false)* is executed to report a refusal to initiate Link Set Up by the remote entity. The second case corresponds to the case where the received frame is not correct or is not a DISC frame. If the received frame is not correct process *isdiscofdm* calls itself; if the received frame is correct but is not a DISC frame either an exit is executed or process *wwaitua* is called. The disable part is the case where the time expires and so another DM frame must be sent.

Issabmofdm Process (776-793)

The first alternative is the case where the received frame is a SABM or a SABME; so an *exit(true, false)* is executed to report an acceptance to initiate Link Set Up by the remote entity. The second alternative corresponds to the case where the received frame is incorrect, or it is not a correct SABM frame or a correct SABME frame.

Isdm process (795-821)

The first alternative is the case where the received frame is a DM frame with final bit equal to 1; so an *exit(false, false)* is executed to report the refusal to initiate Link Set Up by the remote entity. The second alternative is the case where the received frame is not correct or is not a DM frame. The third alternative is the case of collision (i.e reception of DM frame with final bit equal to 0); so process *calloption1* is called to send a SABM or a SABME frame.

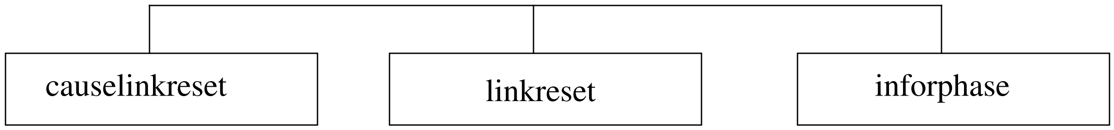
Dataphase Process (1172-2366)

During the information transfer phase (process *inforphase*), reception of particular frames causes the link to be reset (process *linkreset*); also if a frame is sent up to *n2* times without receiving a response the link must be reset. So, process *inforphase* activates process *causelinkreset* which in its turn activates process *linkreset*.

The boolean parameter *notwait* is an input from *connectionphase* process; if it is true an unsolicited response frame does not cause the link to be reset. The parameter *q* is the queue of sent information frames but not unacknowledged yet. The parameter *qdls* is the queue of received information frames built after receiving packets from network layer to be sent to the remote entity.

The parameter *qdlr* is the queue of received packets from remote entity to be delivered to the network layer.

The following figure shows processes composing *dataphase* process:



Causelinkreset Process (1198-1202)

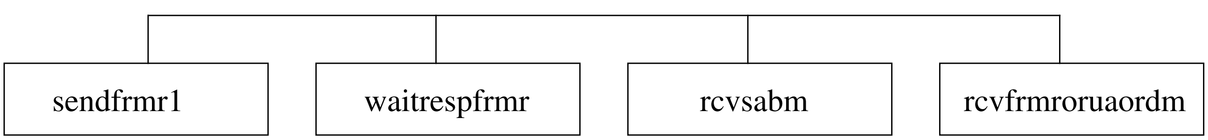
After synchronizing with process *inforphase* in the first action, the synchronization with process linkreset is executed in the second action. The value of the variable *n* distinguishes which kind of linkreset is being done; *reasfrmr* is the field of the FRMR frame in case it is sent.

The variables *vs, vr* and *lu* are respectively the sequence number of the next I frame to send, the sequence number of the expected I frame and the sequence number of the lowest unacked frame.

Linkreset Process (1204-1403)

In case of *n* equal to 0, the *calloption1* process with SABM or SABME as the parameter frame to be sent is called; if Link Set Up is done successfully, Data Phase transfer is resumed with queues *q, qdls* and *qdlr* in the same state as left in the link reset. When *n* is equal to 1 *sendfrmr1* process is called, when *n* is equal to 2 or to 3 process *rcvfrmroruaordm* is called, and when *n* is equal to 4 process *rcvsabm* is called.

The following figure shows processes composing process *linkreset*:



Sendfrmr1 Process (1263-1290)

An FRMR frame is sent then a response frame is waited by process *waitrespfrmr*. An FRMR frame may be sent up to *n2* times; if an FRMR frame is sent *n2* times without receiving a response, Link Set Up is done.

Waitrespfrmr Process (1292-1402)

This process is composed of six alternatives; in the first and the fourth ones, received frames are ignored. The second and the third alternatives are the case where the received frame requires to retransmit the same FRMR frame. The fifth alternative is the case where the received frame is an FRMR; so either a DM frame with final bit equal to 0 is sent to ask the remote entity to initiate Link Set Up, or Link Set Up is done by calling process *calloption1*.

In the sixth case the received frame is either a SABM, a SABME or a DM. If it is a SABM or a SABME frame either a UA frame is sent and Data Phase transfer is resumed, or a DM is sent and Disconnected Phase is entered. If it is a DM frame with final bit equal to0, either a DISC frame is sent and Disconnected Phase is entered, or a SABM or SABME frame is sent by calling *calloption1* process.

Rcvsabm Process (1250-1261)

After receiving a SABM or a SABME frame, either a UA is sent and Data Phase is resumed, or a DM is sent and Disconnected Phase is entered.

Rcvfrmroruaordm Process (1230-1248)

After receiving a DM, UA or FRMR frame, either a DM frame with final bit equal to 0 is sent to ask the remote entity to initiate Link Set Up and the Disconnected Phase is entered, or Link Set Up is done by calling process *calloption1*.

Inforphase Process (1462-2365)

This process is obtained by synchronizing processes *infphase* and *timer* over gate t. Among parameters of *the inforphase* process we find:

-- *vsp*: auxiliary variable used to assign sequence number to packets put in *qdls* queue.

-- *rbusy*: indicates if the remote entity is busy.

-- *lbusy*: indicates if the DTE is busy.

-- *retrans*: indicates if frames in queue *q* must be retransmitted.

-- *chkpt*: indicates if command frame with poll bit was sent (checkpoint) and if so a response frame with final bit equal to 1 must be received.

-- *starttime:* indicates if the timer is running or not.

-- *reject:* indicates that the DTE has sent a REJ frame (after receiving an out of sequence frame) and therefore is expecting an I frame.

-- *init:* counts the number of times a frame has been sent consecutively without receiving a response, if *init* becomes equal to *n2* the link is reset (see process *timeout)*.

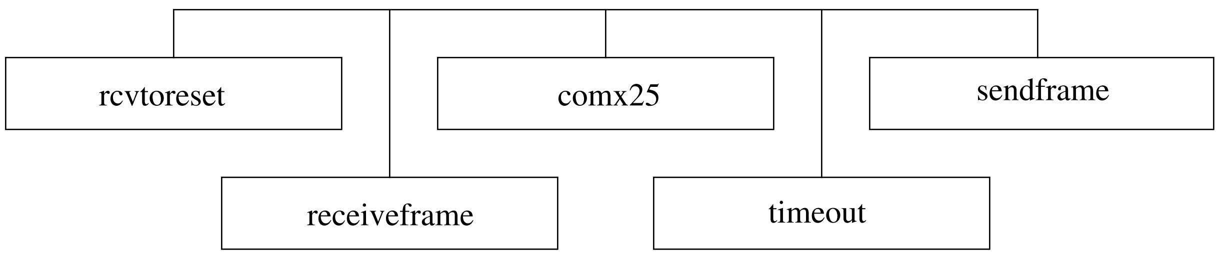
The following figure shows the processes which compose process *inforphase*:

Infphase Process (1472-2364)

During information transfer phase, the DTE can either communicate with the X.25 network layer (process *comX.25*) to receive and deliver packets, send frames to the remote entity (process *sendframe)*, or receive frames from the remote entity (process *receiveframe).* Also , the time out can occur (process timeout). The reception of particular frames which must cause the link to be reset is done by process *rcvtoreset*.

Depending on the value of the variable *typ* one of five alternatives is chosen. The first four alternatives consider the case where the timer must be started or not, or the timer was stopped, so another instantiation of process *timer* is executed; the last alternative considers the case where the link must be reset. Parameter *reject* indicates if an out of sequence frame was received and therefore a REJ frame was sent.

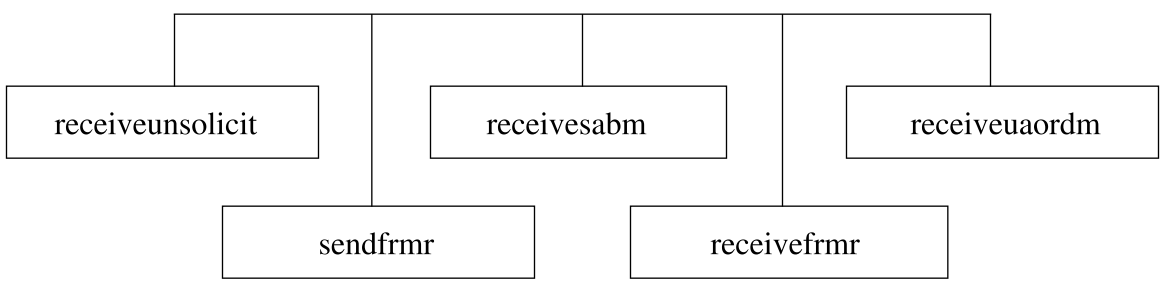
The following figure below shows the processes composing infphase process:



Rcvtoreset Process (1569-1615)

This process causes the link to be reset when receiving a frame which requires so; it is composed of alternatives between processes *receiveunsolicit, receivesabm, receiveuaordm*, *sendfrmr* and *receivefrmr*.

The following figure shows the processes composing process rcvtoreset:



Sendfrmr Process (1405-1460)

This process receives a valid frame which requires an FRMR frame to be sent; such a case can occur in at least one of the following cases:

-- receipt of a frame with an information field which is not permitted or receipt of a S frame with incorrect length.

-- receipt of an undefined command or response control field.

-- receipt of an invalid nr.

-- receipt of an I frame with an information field which exceeds the maximum established length.

These four conditions correspond to the first four alternatives; the interleaving operators are used to determine the *reasfrmr* field of the FRMR frame to be sent.

ComX.25 Process (1523-1530)

This process is used to allow LAPB to communicate with X.25 in order to receive or deliver packets; it is composed by the alternative between processes *recvpack* and *deliver*.

The figure below shows processes composing process *comX.25*:

Recvpack Process (1659-1673)

This process receives packets from X.25 and puts them in queue *qdls* if the number of outstanding frames at the DTE does not exceed the window size. If the remote entity is busy and timer is not running, the variable *typ* is reset to 1 in order to start the timer; else it is reset to 0.

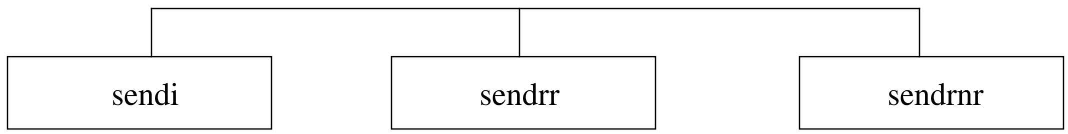
Deliver Process (1617-1626)

This process delivers received packets which are in *the qdlr* queue to the X.25 network layer.

Sendframe Process (1532-1541)

This process is composed by the alternative of processes *sendi, sendrr* and *sendrnr*. It represents the DTE sending a frame to the remote entity. The processes *sendi, sendrr* and *sendrnr* represent respectively the transmission of an I frame, an RR frame and an RNR frame.

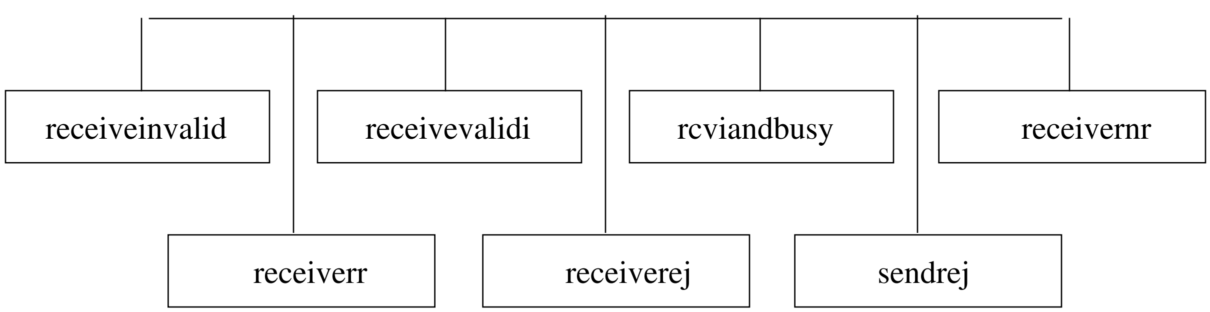
The following figure shows the processes composing process *sendframe.*



Receiveframe Process (1543-1567)

This process is the result of the alternatives between processes *receiveinvalid, receivevalidi, rcviandbusy, receivernr, receiverr, receiverej* and *sendrej.* The process *receiveinvalid* shows that an invalid frame is simply ignored. The process *receivevalidi* represents the reception of a valid I frame. The process *rcviandbusy* represents the reception by the DTE of valid I frame when it is busy, these frames are simply ignored. Processes *receivernr, receiverr* and *receiverej* represent respectively the reception of an RNR frame, an RR frame and a REJ frame. The process *sendrej represents* the reception of an out of sequence I frame and therefore the DTE sends a REJ frame.

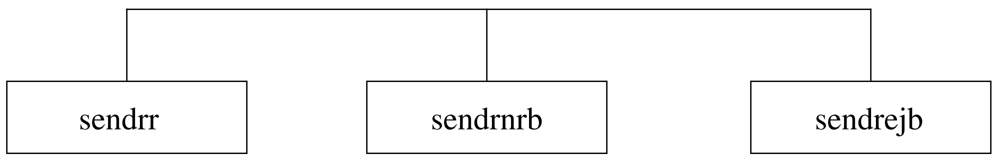
The figure below shows the processes composing process receiveframe:



Timeout Process (2305-2363)

If the timer expires, either an RR command frame, or an RNR command frame with poll bit set is sent if a REJ frame was not previously sent; this is done respectively by processes *sendrrb* and *sendrnrb*. If a REJ frame was previously sent and the timer expires, process *sedrejb* is called to send a REJ command frame with poll bit set. If an RR, RNR or REJ command frame is sent *n2* times without receiving a response frame the variable typ is reset to 5 to reset the link. *Init* parameter indicates the number of times an S command frame was sent without receiving a response.

The figure below shows the processes composing process *timeout*:



Terminationphase Process (845-864)

This process is composed of three alternatives. The first one is the case where a disconnect request is received from X.25, so *calloption1* process is called to send a DISC frame. The second alternative is the case where a DISC frame is received by the DTE, so a UA frame is sent and the Disconnected Phase is entered. The third alternative is the case where the Disconnected Phase is entered due to *dataphase* process in which case *terminationphase* synchronizes with *goterm* process.

Goterm Process (836-843)

This process synchronizes over gate pp with both processes *dataphase* and *terminationphase*.

Chapter 4. Test Case Derivation for LAPB

4.1 Introduction to Conformance Testing

4.1.1 Introduction

Conformance testing involves testing an implementation to establish whether it conforms to the specification in the relevant standard. Conformance testing can be considered as a form of functional testing. In functional testing, the object to be tested is considered as a black-box. It is subjected to inputs and its outputs are verified for conformance to its specification.

Conformance testing consists of testing both the capabilities and the behaviour of the implementation under test (IUT). The capabilities tested are those stated by the implementor. The terms used in the following are defined in the ISO documents [ISO9] and [ISO10].

There are several steps involved in a conformance testing process, which are described briefly as follows [PUH]:

1) test sequence derivation and adaptation: this step consists of obtaining a generic conformance test suite (CTS) for a given protocol. A generic test suite is one that is expressed in a way which is independent of a specific test architecture. Then, the generic CTS is adapted to a particular test architecture and an *abstract* CTS (called ATS for Abstract Test Suite) is obtained.
2) test sequence selection: out of the large number of abstract test cases, an appropriate subset of the abstract test suite is selected.
3) test sequence execution: this step consists of applying an *executable* ATS to the IUT. This assumes a configuration of the environment for running the tests and obtaining the results.
4) test report: this consists of analyzing the test results and producing diagnostic information.

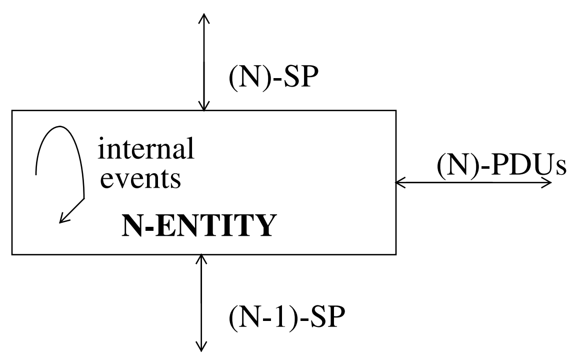
4.1.2 Standards for Conformance Testing

4.1.2.1 Protocol Entity Behaviour

The behaviour of an N-protocol entity can be described in terms of its responses to the:

- requests ((N)-service primitives) from its user ((N+1)-entity).
- messages (protocol data units) from its peer entity ((N)-entity).
- indications ((N-1)-service primitives) from its provider ((N-1)-entity).
- internal events (example: timeouts).

This is shown in the following figure, in which SP means Service Primitive and PDU means Protocol Data Unit:



4.1.2.2 Abstract Test Methods

An abstract test method is described by identifying the points closest to the IUT where control and observation are to be exercised. It represents the testing architecture for each abstract conformance test suite. The OSI protocol standards define allowed behaviour of an N-protocol entity (i.e the dynamic conformance requirements) in terms of the PDUs and both the abstract service primitives (ASP) above and below that entity. Thus the behaviour of an N-entity is defined in terms of the (N)-ASPs and (N-1)-ASPs (the latter including the (N)-PDUs) [ISO9].

The points of control and observation (PCO) are constrained by the architecture of the system under test, which is the system in which the IUT resides. For example, in X.25 DTE testing, it is common for the manufacturer not to provide access to the upper interface of the data link layer [Kan]. The testing entity which provides the means of control and observation of the upper-service boundary of the IUT is called the upper-tester whereas the one which provides the means of control and observation either below or remote from the IUT is called lower-tester.

Currently, ISO has defined five different abstract test methods [Linn], [Ray]. These are defined as follows:

1) **Local Test Method (L)**: in this method, the PCOs are at the local service boundaries above and below the IUT (figure 4.1.a)

2) **Distributed Test Method (D)**: in this method, the PCOs are distributed in two different locations. They are at the service boundary above the IUT and at the opposite side of the (N-1) service provider for the (N)-entity (figure 4.1.b). The test events are specified in terms of the (N-1)-ASPs, (N)-PDUs and (N)-ASPs. A test coordination procedure (TCP) is used to coordinate test execution actions between the upper tester and the lower tester. The requirements for the TCP are specified in the ATS.

3) **Coordinated Test Method (C)**: this method is an enhanced version of the D method. There is a single PCO at the point of interface between the lower tester and the underlying service provider (figure 4.1.c). The TCPs are implemented as a test management protocol, which is expressed in terms of TM-PDUs. The test events are specified in terms of (N-1)-ASPs, (N)-PDUs and TM-PDUs. The requirements for the TCPs are specified in the ATS. The upper tester interface to the IUT is inaccessible. The upper tester is usually implemented by the IUT manufacturer.

4) **Remote Test Method (R)**: in this method, the PCO is at the interface between the lower tester and the (N-1)-service provider (figure 4.1.d). The test events are specified in terms of (N-1)-ASPs and (N)-PDUs. Some of the requirements for the TCP may be implied or informally expressed in the ATS, but their realization is implementation dependent.

5) **Relay Test Method (Y)**: this method is applicable while testing relay systems from one subnetwork to another.

In each of these five methods, testing can be performed for a single layer(S), multiple layers (M) or a single-layer embedded (E) [Ray]. The single layer testing is applicable for single layer IUTs. In the multi-layer testing, the behaviour of a multi-layer IUT is tested as a whole rather than tested layer by layer. The single-layer embedded testing consists of testing the behaviour of a single layer within a multi-layer IUT without accessing the layer boundaries for that layer within the IUT.

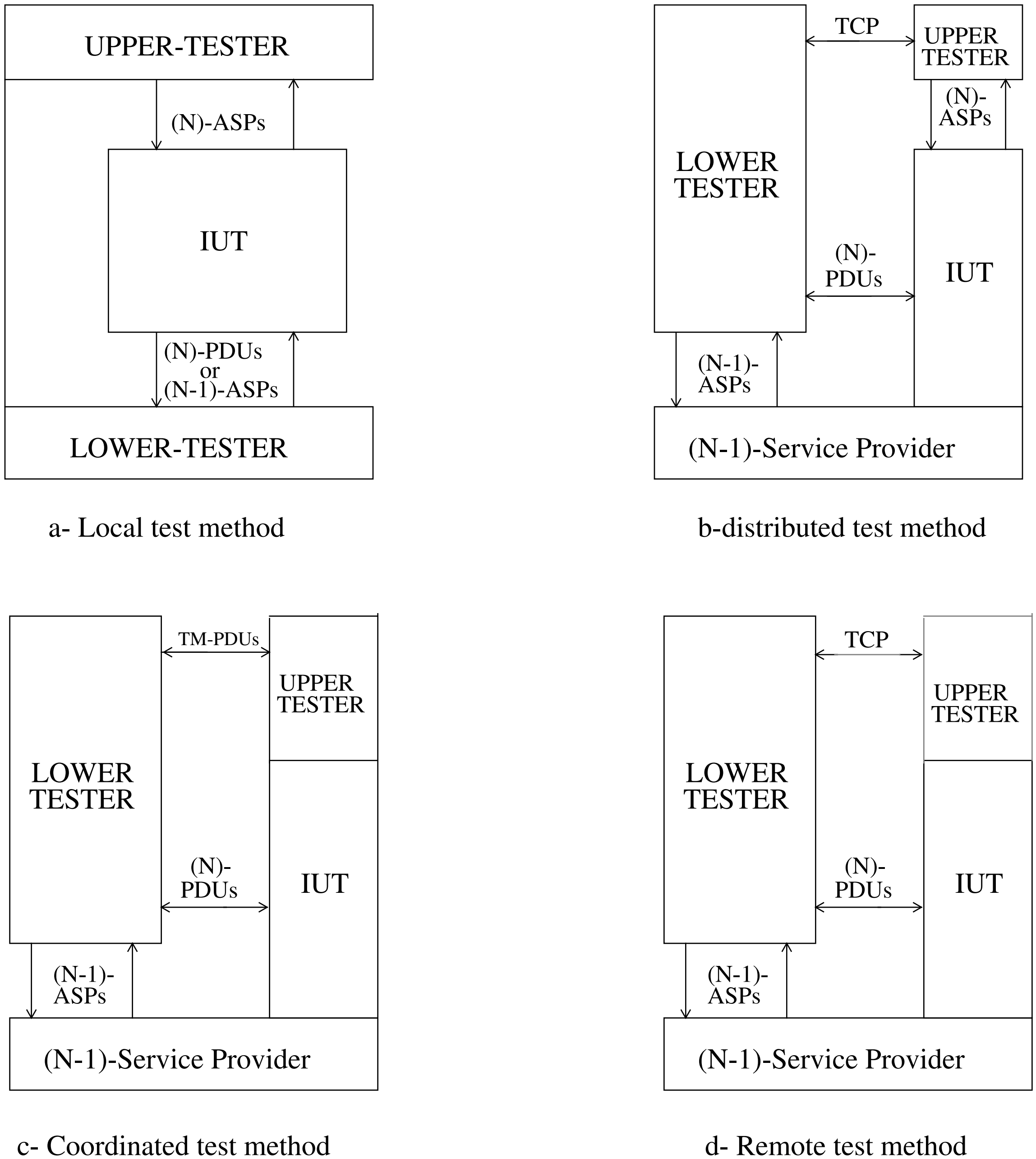


Figure 4.1 - Abstract Test Methods

4.1.3 Conformance Requirements, PICS and PIXIT

The conformance requirements in a standard can be [ISO9]:

a) mandatory requirements: these are to be observed in all cases.

b) conditional requirements: these are to be observed if the conditions set out in the standard apply.

c) options: these can be selected to suit the implementation, provided that any requirements applicable to the options are observed.

Furthermore, conformance requirements in a standard can be specified either positively stating what shall be done or negatively stating what shall not be done (prohibitions). Also, conformance requirements fall into two groups: static conformance requirements and dynamic conformance requirements.

The static conformance requirements are those requirements that define the allowed minimum capabilities of an implementation, in order to facilitate interworking. These requirements may be at broad level, such as the grouping of functional units and options into protocol classes, or at a detailed level, such as a range of values that has to be supported for specific parameters or timers. All capabilities not explicitly stated as static conformance requirements are to be regarded as optional [ISO9].

The dynamic conformance requirements are those requirements which determine what observable behaviour is permitted by the relevant OSI standard in instances of communication. They define the set of allowable behaviours of an implementation. This set defines the maximum capability that a conforming implementation can have within the terms of the OSI standard [ISO9].

To evaluate the conformance of a particular implementation, the implementation manufacturer provides a Protocol Implementation Conformance Statement (PICS). The PICS is a set of statements made by the supplier of an OSI implementation stating the capabilities and options which have been implemented, and any features which have been omitted [ISO9]. First, the PICS is analyzed during the static conformance review for its consistency with the static conformance requirements specified in the relevant standard. Then, the PICS is used, during the test selection step, to select the tests which are appropriate to optional features. In addition to providing the PICS, the supplier provides a Protocol Implementation eXtra Information for Testing (PIXIT). The PIXIT contains information needed by the test laboratory in order to be able to run the appropriate test suite. The PICS and PIXIT proformas of the LAPB protocol can be found in [ISO4].

4.1.4 The Test Specification Notation-TTCN

In principle FDTs are good candidates as test notations [BDS]. In particular, it is shown in [Steen] that LOTOS fulfills the most important requirements of a test notation. This may be controversial, but in the experience of the author, LOTOS may be adequate for specifying test cases. The only problem could be a few-time related features, such as statements TIMEOUT and ELAPSE in TTCN, which are not provided in LOTOS. An advantage of using an FDT as a test notation is that no translation is required if test cases are derived from a formal specification or are validated against a formal specification.

However, ISO and CCITT normalization groups have developed a new language called TTCN (Tree and Tabular Combined Notation) and have adopted it as a standard for specifying tests of conformance [ISO6]. Since TTCN is a standard notation and has been used for specifying test cases for the data link layer in [ISO4], we have adopted it in our thesis to specify test cases.

TTCN is an informal notation with clearly but not formally defined semantics [ISO6]. TTCN is provided in two forms: a graphical representation (TTCN/GR) and a machine processable representation (TTCN/MP). The graphical form is suitable for human readability. The machine processable form is suitable for transmission of TTCN descriptions between machines.

TTCN serves the following purposes:

-- to provide a common reference for assessing other notations and to assist in examining the problems arising in test and test suite design.

-- to provide a basis for the translation of tests into other notations.

-- to be used as a tool to develop trial tests.

A test suite can be looked upon as a hierarchy ranging from the complete test suite down to test events. A test suite contains test groups which contain test sub-groups, containing test cases, which have test steps which include test events. A test suite written in TTCN has four parts: *a test suite overview part*, *a declaration part*, *a dynamic part* and *constraint part*. These are briefly described in the following. More details can be found in [ISO6].

1) Test suite overview: this section shall include at least the following information:

-- a precise indication of the test method to which the test suite applies.

-- an overview and list of contents of the test suite by test subgroups and test group.

-- a complete index to the test suite.

2) Declaration part: the purpose of this part is to describe the set of test events and all components in the test suite. These include ASPs which occur at PCOs, timer identifiers, user defined types and operators, test suite parameters, global variables and constants, PCOs, PDUs and their

parameters and any abbreviation used in the test suite specification. Proformas are available for each of these items. The figure 4.2 shows an example of data type declaration.

DATA TYPE DECLARATION		
PDU: DISC	COMMENTS: None	
Protocol Control Information		
FIELD NAME	TYPE	COMMENTS
Address field (A)	Octetstring	As defined in X.25 standards
Poll bit (P)	Bitstring	Single Bit defined within frame control field defined in X.25 standards

Figure 4.2 - Data Type Declaration Example

3) Dynamic part: this part contains the main body of the test suite. It consists of dynamic behaviour tables which contain columns for *behaviour descriptions, labels, constraints to particular ASPs or PDUs, verdicts* and *comments*. The purpose of the test case, as well as an identifier and a reference to the test case are also specified in a dynamic behaviour table. The field *extended comments* is provided for general comments and the field *default references* is used to emphasize a set of 'interesting' paths through a test by declaring the less interesting common alternatives as default behaviour.

In the behaviour tree a set of alternatives is expressed by writing the first symbol of each alternative aligned in the same column (level of indentation). The events initiated to named PCOs (sent events) are identified by the symbol '!', and test events accepted (received events) at named PCOs are identified by the symbol '?'. Synchronization constraints can be used to synchronize events at distinct PCOs. Other test events used in the behaviour tree are pseudo-events, such as OTHERWISE, TIMEOUT and ELAPSE. OTHERWISE denotes the reception of any event which is not explicitly listed among the alternatives preceding the OTHERWISE. TIMEOUT is used to check the expiration of the specified timer. ELAPSE provides a means of checking how long the execution of a test case remains cycling around a set of alternatives without any of them matching and exiting from that set of alternatives if the prescribed time has elapsed.

Test events may be associated with expressions. There are two kinds of expressions: assignment expressions and boolean expressions. An event is qualified by a boolean expression when that expression is written between square brackets. An event contains an assignment expression when that expression is written between parentheses. A set of operations, such as START and CANCEL, are used to model the timer management. Trees may be attached to other trees by inserting the name of the tree to be attached prefixed by the symbol '+'. A label may be placed in the label column for an event. A jump to a label 'A' is specified by the notation: -> A or GOTO A

The repeat statement is a mechanism allowing to iterate a test step a variable number of times. The constraint reference column allows reference to be made to a specific constraint placed on an ASP or a PDU. Such constraints are defined in the constraint part. Entries in the verdict column indicate a verdict of the test case. Three kind of verdicts are available:

- -- Pass: some aspect of the test purpose has been achieved.
- -- Inconclusive: something has occurred which makes the test case inconclusive for some aspect of the test case
- -- Fail: a protocol error has occurred or some aspect of the test purpose has resulted in failure.

The figure 4.3 shows an example of a dynamic behaviour table for the LAPB DTE, where the TTCN subtrees DL1_state and DL1_CHK are assumed to be already defined.

.

DYNAMIC BEHAVIOUR				
REFERENCE: X.25-DTE/DL1_101 IDENTIFIER: DL1_101 PURPOSE: Verify that the DTE sends a DM/F=1 in response to DISC/P=1 in the Disconnected State DEFAULT REFERENCE: None				
BEHAVIOUR DESCRIPTION	LABEL	CONSTRAINT REFERENCES	VERDICT	COMMENTS
DL1_101				
+DL1_STATE				
L ! DISC (P:=1)		DISC(S31)		
L? DM [F=1]		DM(S31)		
+DL1_CHK				
L? OTHERWISE			FAIL	Unacceptable response
ELAPSE TM01			FAIL	No response
Extended Comments: None				

Figure 4.3 - Dynamic Behaviour Example

4) constraint part: this part allows the description of the value coding of parameters in ASPs and PDUs. TTCN provides proformas for such specifications. Specific binary or hexadecimal values are assigned to ASPs and PDUs parameters. Figure 4.3 shows an example where the constraint reference DISC (S31) is referenced in the dynamic behaviour. This constraint reference is defined in the PDU constraint part as shown in figure 4.4. This figure shows the definition of the contents of the fields A and P.

PDU Constraints			
PDU Name: DISC			
LIST NAME	FIELD NAME		COMMENTS
	A	P	
S31	'01'H	'1'B	

Figure 4.4 - PDU Constraint Example

4.1.5 The Test Method for the LAPB DTE

Usually, the DTE manufacturer does not provide access to the upper boundary of the X.25 DTE data link layer. Therefore, the local and the distributed test methods cannot be used. The coordinated test method requires the existence of an upper tester. This is costly, since the upper tester is used only for testing and once the IUT has been tested, the upper tester is removed. Also, the manufacturer could have implemented level two and three in a single module where they cannot be separated, so it would be impossible to conceive an upper tester for level two only. This has led us to consider the remote test method, as in [Kan]. This is similar to the architecture shown in figure 4.1.d, but it is not exactly the same because the X.25 standard is not exactly aligned with the OSI architecture.

In the remote test method some requirements for test coordination procedures may be implied or informally expressed in the abstract test suite but no assumption is made regarding their feasibility or realization [ISO9].

In the test implementation and execution of the X.25 link layer test suite, the correctness of behaviour is judged solely on PDU exchanges at the DTE-DCE interface; this corresponds to the

(N)-PDUs exchange of figure 4.1.d. There is only one PCO; this PCO (called L in 4.5.4) corresponds to the gate phl (figure 3.1). Some test cases involve initiated actions by the user layers above the IUT. For example, to test that the IUT retransmits an I frame after receiving a REJ frame, an I frame must be first transmitted by the DTE. To test such a case involves an action to be initiated by the DTE.

This can be done by, simply, making a call to a DTE operator asking him to perform the required action, when a test case requires an initiated action. It can be done better, if the test tool provides a user-prompting terminal which helps coordinate DTE-initiated actions. When a test case requires the IUT to initiate an action, the test tool writes a user-prompt message at the remote terminal. The DTE operator performs the requested action and acknowledges the prompt by sending a response message. This message exchange corresponds, roughly, to the TCP of figure 4.1.d. Figure 4.5 shows a test architecture for X.25 testing, developed in collaboration between Bell-Northern Research (BNR) and the university of Ottawa [PUH]. The network test system (NTS) provides all the facilities for developing and executing X.25 test cases. For each particular protocol IUT, there is an interactive protocol tester (IPT). The IPT emulates PDU encoding and decoding functions for a particular protocol and provides exception and/or error generation and handling for that protocol.

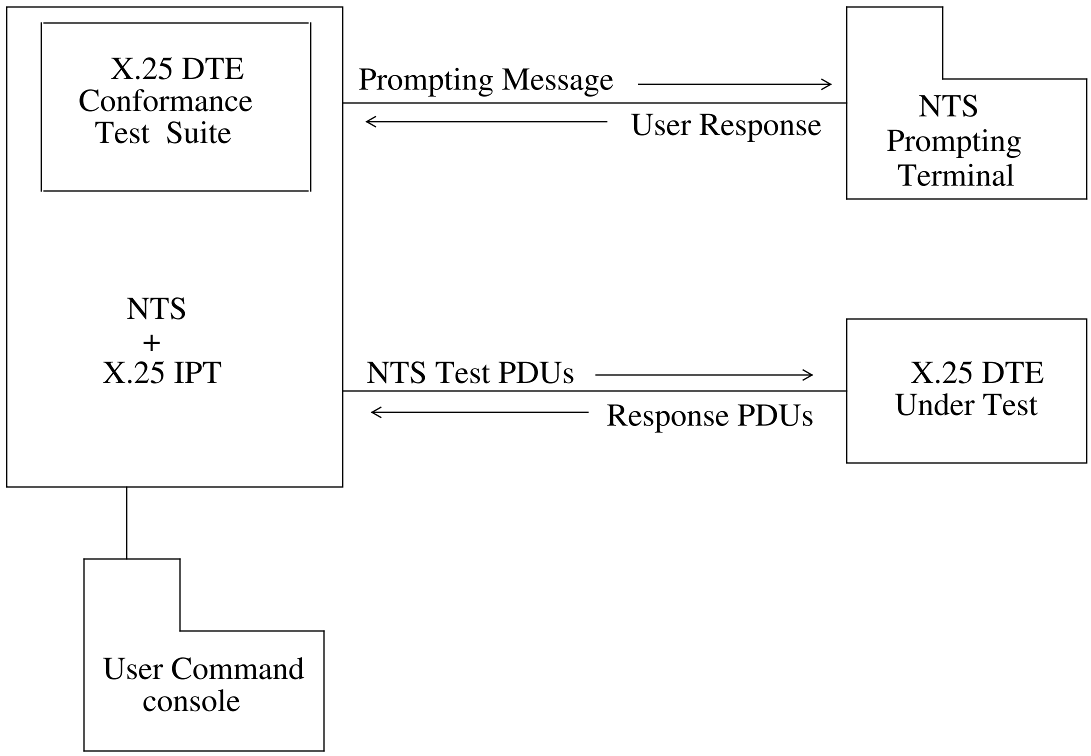


Figure 4.5 - A Test Architecture for X.25

4.2. FSM Test Sequence Derivation Methodologies

4.2.1 Introduction

In FSM test sequence derivation methodologies the communication protocol is specified using the FSM approach where the system is modeled using states and transitions. Given a protocol specified by a FSM there are several methods for generating test sequences from this FSM. All these methods assume a deterministic FSM. The ability of a test sequence to decide whether a protocol implementation conforms to its specification relies solely upon the range of faults that can be captured. Also, the length of generated test sequences is important. The D, U and W method use the synchronizing sequence (SS) to bring the FSM to a specified initial state in order to perform state identification and transition verification. A SS of a FSM is an input sequence that takes the FSM to a special initial state regardless of the output or the initial state of the machine. We review briefly below the different FSM test sequence derivation methodologies. We use the following notation:

TR: transfer sequence which takes the machine from one state to another.

T: transition.

UIOi: a UIO for state i.

4.2.2 The Transition Tour Method

Starting with the initial state, a test sequence (called transition tour sequence), is applied to the FSM until the machine has traversed every transition at least once. This method assumes a minimal, strongly connected and completely specified machine [SL]. It detects only output errors of transitions.

4.2.3 The D Method

This method is designed for machines that have a distinguishing sequence (DS). A DS is an input sequence for which the output sequence produced by the machine is different for each initial state. This method assumes a FSM which is minimal, strongly connected, completely specified and posses a DS [SL]. It detects output and tail state errors of transitions.

A state identification is performed by the sequence: SS-TR-DS

A transition verification is performed by the sequence: SS-TR-T-DS

4.2.4 The U Method

This method involves deriving a unique input/output (UIO) sequence for each state of the machine. A UIO sequence for a state of the machine is an input/output behaviour that is not exhibited by any other state of the machine. This method assumes a minimal, strongly connected and completely specified machine [SL]. It detects output and tail state errors of transitions.

A state identification is performed by the sequence: SS-TR-UIOi

A transition verification is performed by the sequence: SS-TR-T-UIOi

4.2.5 The W Method

This method is based on the use of a characterizing set of a FSM. A characterizing set W of a FSM M is a set of input strings $w_1, \ldots, w_k$ such that the last output symbols observed from the application of these strings (in a fixed order in order to compare between the different output symbols obtained) are different at each state of the FSM, i.e $Ms_1(w_1, \ldots, w_k) \neq Ms_2(w_1, \ldots, w_k)$ where s_1 and s_2 are any two different states and $Ms_i(w_1, \ldots, w_k) = (Ms_i(w_1), \ldots, Ms_i(w_k))$, i=1,2. Ms denotes the machine at state s and $Ms(w_i)$ denotes the last output symbol on input string w_i to Ms. This method assumes a minimal, strongly connected and completely specified machine [SL]. It detects output and tail state errors of transitions.

A state identification is performed by the sequences:

$$
\begin{gathered}
SS\text{-}TR\text{-}w_1 \\
SS\text{-}TR\text{-}w_2 \\
. \\
. \\
SS\text{-}TR\text{-}w_k
\end{gathered}
$$

A transition verification is performed by the sequences:

$$
\begin{gathered}
SS\text{-}TR\text{-}T\text{-}w_1 \\
SS\text{-}TR\text{-}T\text{-}w_2 \\
. \\
. \\
SS\text{-}TR\text{-}T\text{-}w_k
\end{gathered}
$$

4.2.6 Conclusion

All four previous methods assume that a minimal, strongly connected and fully specified machine of a protocol entity is given. Except for the D method, the other three methods guarantee the existence of a testing sequence for a FSM which satisfies the assumption of minimality, strong connectivity and complete specification [SL]. For the U method, a complete specification of a FSM is only a sufficient condition for producing a set of UIO sequences whereas it is a necessary condition for generating a DS or a W set. As a general evaluation of the four methods, the following comments have been made in [SL]: the T method is simple but may not capture all single faults if they actually exist; D-method is more involved in its implementation and requires the existence of a DS, which may not exist for a FSM; U-method is easy to comprehend; and W-method, in general, will produce longer test sequences than others.

Our methodology of generating test cases for the LOTOS LAPB specification is based on the use of the UIO method. The main idea behind this is that LOTOS can be seen as a labelled transition system. Labelled transition systems are a generalization of finite state machines. Therefore, we thought of using FSM test sequence derivation methods for our LAPB LOTOS specification, since there is no general method for generating test sequences from LOTOS specifications, up to now. For its simplicity, its fault coverage and the length of produced test sequences, we choose the UIO method. Our LAPB specification was written to be complete but there was no guarantee that UIO sequences exist because of the non-determinism involved in the LAPB behaviour. Fortunately, we found that UIO sequences exist.

4.3 Test Sequence Derivation for Basic LOTOS

4.3.1 Terminology

Before giving the main idea of a methodology of derivation of test sequences for a restricted subset of LOTOS [Brink1], we define the following terms:

Basic LOTOS: a simplified version of the language where the observable actions are identified only by the names of the gates. Therefore, no data is involved in basic LOTOS.

Full LOTOS: the structure of actions is enriched by allowing the association of data values to gate names.

Nondeterminism: an example of nondeterminism is represented inside process *called* in the LAPB specification as shown below:

```
 phl ! SABM
;( i; UA
  []
   i;DM )
```

After receiving a connection setup request (SABM frame), the DTE may accept this request by sending a UA frame or may refuse this request by sending a DM frame. This is an internal decision of the DTE and therefore is unobservable by the environment. In this case the nondeterminism is modeled by the nonobservable action *i.*

Deterministic process: a process is *deterministic* if no "nondeterminism" exists in this process.

Reduction [Brink1]: a process I reduces a process S if and only if:

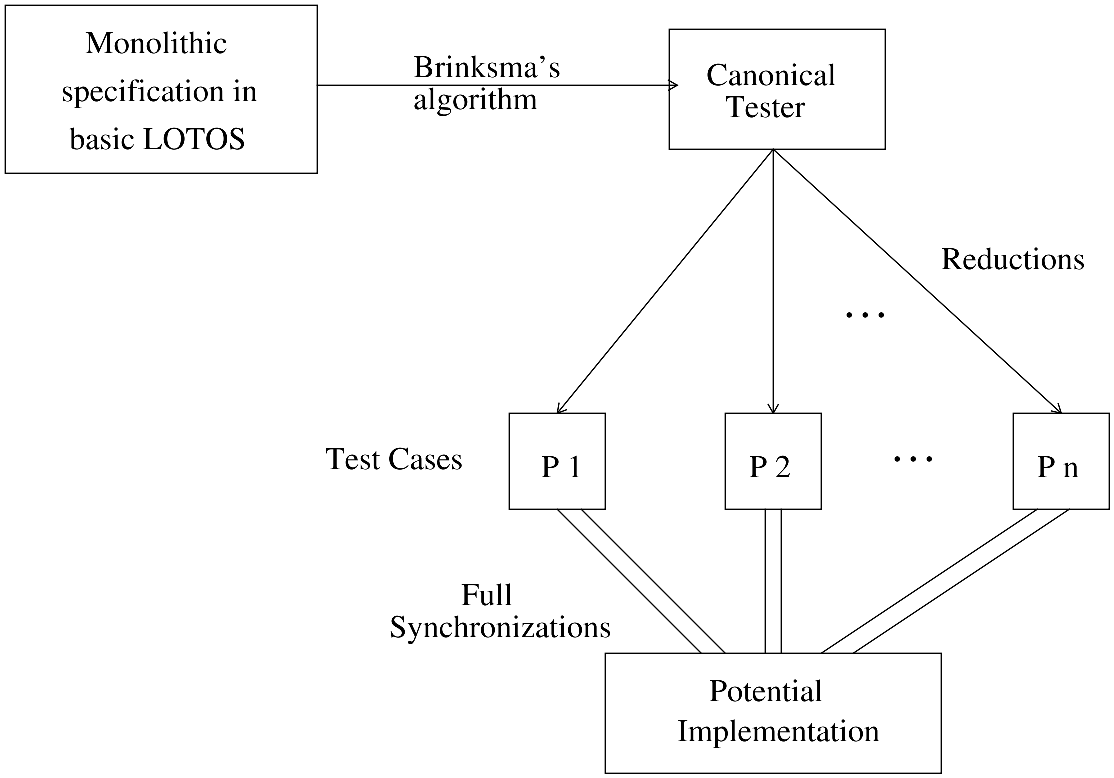
(i) I can only refuse actions that can be refused by S.

(ii) I can only execute actions that can be executed by S.

Point (i) reflects the conformance of I to S; i.e any sequence of actions accepted by I must be accepted by S. Point (ii) signifies that I cannot extend the capabilities of S. Informally, I can be said to be an "implementation" of S.

4.3.2 Main Idea

Given a monolithic specification in basic LOTOS, tests can be generated from this specification to allow checking the consistency of a potential implementation with respect to the specification. First from the monolithic specification written in basic LOTOS what is called the canonical tester of this specification is constructed using the algorithm described in [Brink1]. Then by doing reductions on this canonical tester, deterministic processes which are the test cases are obtained. This is shown in the following figure:



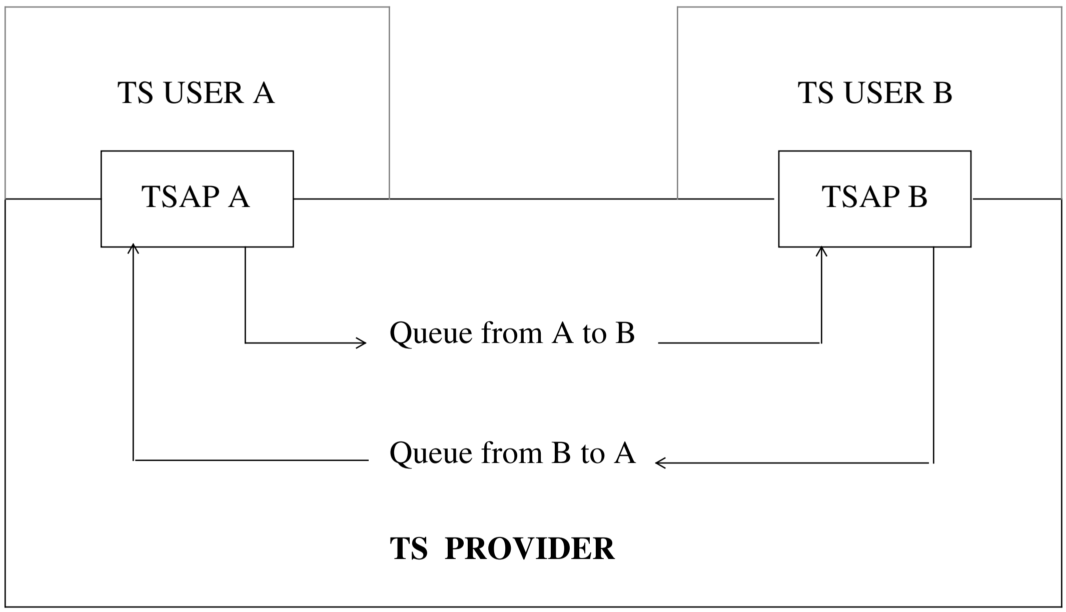
An implementation conforms to the specification if no one of the obtained deterministic processes when composed in full synchronization with the given implementation leads to a deadlock before the deterministic process itself reaches a deadlock. In other words, if the full synchronization of any one of the deterministic processes with the given implementation reaches a deadlock before the deterministic process alone reaches a deadlock, we can definitely say that this implementation does not conform to the specification.

Notice that test cases are not generated from the specification but from its canonical tester. This latter, which is a LOTOS process, can be seen as an extension of the specification and as a specification of an adequate test suite. The implementation is also seen as a LOTOS process. Brinksma proves that any specification in basic LOTOS without recursion has a canonical tester [Brink1].

Example

An example of the first half Connection Phase (i.e up to and including connection indication or disconnection primitives) of a single connection of a transport service (TS) illustrates the methodology of generating test sequences from a specification written in basic LOTOS.

The operation of a transport connection may be represented as shown in figure below. This example is due to Brinksma [Brink1].



There is one queue for each direction of information flow. A pair of queues exists for each transport connection existing between two users. The TS access points are a resource shared by the TS users and the TS provider; this is where the interactions or primitives occur.

The service users are restricted to only two; one acting as a calling user and the other as a called user. The address at which a primitive occurs is indicated by extending the name of the primitive with _A for the calling user and with _B for the called user.

Process S shown in the following characterizes such a first half Connection Phase of a TS. Note that the *exit* leads to either the completion of the Connection Phase or to the release of the phase.

```
process S := ConReq_A;
        ( ConInd_B;exit
         []
          DisReq_A;
          ( i;exit
```

```
                  []
                    ConInd_B;exit
                  )
                []
                  (i;DisInd_A;
                   ( i;exit
                   []
                     ConInd_B;exit
                   )
                  )
endproc
```

Applying Brinksma's algorithm it turns out that the canonical tester T(S) of S is actually the same as S. Then applying reductions to T(S) we arrive at the following four test cases:

(1) ConReq_A; DisInd_A; exit(pass)
(2) ConReq_A; DisInd_A(exit(pass)[]ConInd_B; exit(pass))
(3) ConReq_A; (DisInd_A; exit(inconc)[]DisReq_A; (exit(pass)[]ConInd_B; exit(pass))
(4) ConReq_A; (DisInd_A; exit(inconc)[]ConInd_B; exit(pass))

Case(1) tests the fact that a connection can always be refused by the service provider and actually will be refused if the service provider is not given the opportunity to execute a connection indication primitive. Case(2) tests that after a connection is refused a connection indication at B may or may not be performed.

Case(3) tests the case where a Disconnection request at A occurs after a connection request at A was performed. Notice that since the purpose of the test case is to test that a DisReq_A occurs after ConReq_A and since S can engage in action DisInd_A (due to nondeterminism), an inconclusive verdict is issued. Case(4) tests the case where a connection indication at B occurs as a result of the connection request at A. For the same reason as previously, an inconclusive verdict is issued if S engages in action DisInd_A.

Note that the verdict *pass* and *inconclusive* are not provided by this testing methodology. Rather, they are formulated by a testing expert on the basis of his insight on the purpose of the test.

Another underlying assumption is that all sequences of actions not mentioned above lead to verdict *fail*.

4.3.3 Practical Limitations

As mentioned above the methodology explained previously can be applied only to finite monolithic specifications written in basic LOTOS; obviously real data communications protocols are too complex to be expressed by such specifications. The most important limitation is that this methodology does not consider data. This is due to the fact that the basic idea for the construction of the canonical tester is very close to the failure model of CSP; but when dealing with full LOTOS, there are guards and selection predicates that cannot be evaluated statically.

Also, the recursion results in infinite processes. When dealing with large specifications, it may become impractical to derive tests on the basis of the whole specification; the possibility to decompose the specification into smaller parts and then generate test cases for each part can reduce the problem but then one has to find out how to decompose a specification. On the other hand, in practice tests are only possible on finite portions of the behaviour of an entity.

The last limitation is that this methodology requires a synchronous communication between the tester and the implementation under test (IUT) which implies a very strong control over the behaviour of the IUT; it would be interesting to see how to extend this to nonsynchronous interfaces by assuming asynchronous interactions between the tester and the IUT.

4.4 Test Sequence Generation for a LAPB FSM Specification

4.4.1 Introduction

The LAPB DTE operation is specified as a FSM. In order to avoid dealing with a large FSM, states are grouped according to protocol phases.

As a first step the state transition diagram for the LAPB protocol is defined. The state transition diagram specifies the behaviour of the DTE at the data link level. Next, a validation tool to check the connectivity and completeness properties of this FSM is used. Validation tools are based on reachability analysis of the state machine or Petri net representation. Reachability analysis is performed in order to validate the protocol against potential errors like state deadlocks, unspecified receptions and nonexecutable interactions.

The third step is to specify the state transition table. Seven states are defined to describe the DTE's operation. For each state, the appropriate actions and the next state after taking these actions for the various proper, improper and inopportune PDU's are shown in the transition table.

The last step is to derive test cases using the state transition table. This is done using a three step procedure as explained in section 4.4.3.

4.4.2 The FSM of LAPB

The figures 4.6.a, 4.6.b and 4.6.c show the FSM of the LAPB DTE. The transitions may be enabled by an enabling predicate. For example, the predicate (V(S) <= N(R) + K) represents the condition that the send state variable is within the window. Similarly, transitions in the event of time out (T1) are expressed with a suitable enabling predicate. The data link layer uses command and response PDUs. These are distinguished by using underlined or normal type font, respectively. The '?' symbol is used to denote a reception and '!' symbol is used for transmission. When neither is specified this denotes both a reception and a transmission.

The DCE states are not shown separately, i.e, only the DTE states are shown. The DCE in this case is the tester. The tester follows the DCE state in the 'ideal case', that is, for generating proper PDUs. Note that improper and inopportune PDUs are not shown in the state transition diagrams. However, they are part of the input set from the tester to the (IUT). The expected output from the IUT are in the set of all proper PDUs. Reception at the tester of any PDU outside the range of the expected outputs correspond to a test failure for the IUT. However, the tester can send improper PDUs to the IUT to verify its reaction to such PDUs.

4.4.3 Test Sequence Derivation Method

The test cases are generated in TTCN. The state transition table represents the valid behaviour of the IUT for a given PDU sent from the tester. Test cases are derived from the state transition table using a three step procedure:

(1) test initialization step.

(2) evaluation step.

(3) verification step.

The *test initialization step* consists of bringing the DTE to the required state. This step assumes that the DTE's FSM is strongly connected. The figure 4.7 shows the TTCN subtree init_disconnect which initializes the DTE to the Disconnected State:

```
Init_disconnect
L ! DISC ( P:=1 )
     L ? UA [ F=1 ]
     L ? DM [ F=1 ]
     L ? Otherwise            fail
     ?Elapse  Td              fail
```

figure 4.7 - TTCN Initialization Subtree for the Disconnected State

The evaluation step consists of performing a simple transition test or window rotation test. During the evaluation test step the input sequence to the IUT can be from the set of proper, improper or inopportune PDUs.

The evaluation step consists of a single interaction; this interaction consists of:

-- sending a PDU as a test stimulus.

-- verifying the acceptable response including no response if allowed in the state transition table.

The verification step consists of verifying the IUT's state after the transition performed in the evaluation step.

Most test cases consist of a preamble followed by a test body and ending with a postamble. The preamble consists of the initialization test step. The test body consists of the evaluation step. The postamble consists of the verification step. The following example illustrates an abstract test case in TTCN:

```
+preamble
   +test_body_part 1
       +postamble 1
   +test_body_part 2
       +postamble 2
     .
     .
```

.

+test_body_part n

+postamble n

The ’+’ symbol indicates the attachment of a TTCN tree. Trees attached at the same level of identation are alternatives.

4.4.4 Discussion of Results

The test cases are divided in eight groups [ISO4], [Kan]. Seven of these groups are provided to test the interaction capability of the DTE in every phase. The eighth test group tests the setting of the following system parameters:

-- T1: retransmission time recovery.

-- N2: maximum number of attempts made to complete a transmission.

The test groups are further divided in three subgroups; these are as follows:

-- subgroup 1: involves those test cases where the tester transmits a proper test frame.

-- subgroup 2: involves those test cases where the tester transmits improper test frames.

-- subgroup 3: involves those test cases where the tester transmits inopportune test frames

Seven test subtrees, each one representing the preamble of one of the seven DTE states, are defined as a part of a common library of test steps. A preamble allows to bring the DTE to an initial test state from any state of the DTE. In the case of the Disconnected Phase, this is done simply by sending a DISC frame from the tester and then receiving a UA or a DM response. In the case of the other states, the DTE is first brought to the Disconnected Phase and then taken to the appropriate state.

Seven other test subtrees, each one representing the postamble of one of the seven DTE’s states, are defined as part of the common library of test steps. A postamble allows to verify that the DTE is at the expected state.

4.5 Test Sequence Derivation for a LAPB LOTOS Specification

4.5.1 The Symbolic Tree

Given a process, the LOTOS interpreter is able to systematically generate a labeled symbolic tree (LST) up to given maximum lengths and widths. Such a tree shows all possible execution sequences for the entity specified by the process. 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 are simply ignored. As an example, the symbolic tree with a length equal to four of process P, shown below, is given in the following:

```
process P[a,b] : noexit :=
     a; b; P[a,b]
    []
     b; a; P[a,b]
endproc
```

```
1 a
| 1 b
| | 1 a
| | | 1 b  ==> continue
| | 2 b
| | | 1 a  ==> continue
2 b
| 1 a
| | 1 a
| | | 1 b  ==> continue
| | 2 b
| | | 1 a  ==> continue
```

4.5.2 Limitation of The Symbolic Tree

4.5.2.1 Introduction

When obtaining the symbolic tree of a process, the interpreter generates a great number of sequences that are either infeasible, in the sense that they relate to logically impossible paths, or redundant, in the sense that they differ the ones from the others by the placement of nonrelevant internal events.

Of course eliminating all impossible paths and removing all nonrelevant internal events is an undecidable problem. Therefore, these execution sequences are simplified by using heuristics in order to make them useful for testing purposes by constructing a *simplified symbolic tree* (SST).

4.5.2.2 Obtaining Finite Trees

The trees obtained are usually infinite. This is the normal case when recursion is involved. Two methods for coping with infinite paths are detecting recursion, and ignoring some paths under user control to construct the SST.

Detecting Recursion

Repeated occurrence of the same behaviour expression along a tree path is a case of recursion that can be detected automatically. A unique identifier is associated with an occurrence of a behaviour expression in a tree. Later, occurrences of the same behaviour expression in the same tree are replaced by the identifier preceded by the word '*again*'. As an example, the LST and the SST of process P are shown below.

The LST is:

```
1 a
| 1 a
| | 1 a
| | | ...
| | |
| | | ...
| | 2 b
| | | 1 exit ** EXIT SUCCEED **
| 2 b
| | 1 exit ** EXIT SUCCEED **
2 b
```

| 1 exit ** EXIT SUCCEED **

While the SST is:

bh0 * 1 a ==> again bh0
* 2 b
bh1 * | 1 exit ** EXIT SUCCEED

Ignoring Some Paths

In generating behaviour trees for complex systems, it is normal that one may wish to ignore certain paths. For example for paths relating to the creation of several connections, it can be useful to consider the case of one connection only. Such paths are usually preceded by internal actions; so one is allowed to specify that the entire subtree following a certain internal action has to be ignored.

4.5.2.3 Infeasible Paths

One may be able to eliminate certain paths that are not feasible by trying to evaluate symbolically guards and selection predicates. Predicates that cannot be evaluated to false and which are not in contradiction with others previously assumed to be true, are assumed to be true.

The detection of contradictions is in general an undecidable problem. Some heuristics are used. Contradictions such as *(q(x) and p(x))*, where *q(x)=not(p(x))* appearing in the list of axioms are detected automatically. In addition the user can establish a data base of contradictions.

An example is:

psabm[phl]
|[phl]|
pdisc[phl]

where processes psabm and pdisc are:

process Psabm[phl] : **exit** :=
(phl ? f: frame [is-sabm(f)]
; phl ! UA; **exit**)

```
 []
   phl ? f:frame [not(is-sabm(f))]
   ;phl ? f: frame;exit
endproc

process Pdisc[phl] : exit :=
  (phl ? f: frame [is-disc(f))]
   ;phl ! UA; exit)
 []
  phl ? f: frame [not(is-disc(f))]
  ;phl ? f: frame; exit
endproc
```

Assume that the data base of contradictions contains:

[is_sabm(f)]#[is_disc(f)]

This expresses the simple fact that a SABM frame cannot be a disconnection frame. Suppose that the maximum length has been set to two, then before simplification, the tree is:

```
1 phl ?[f,f]frame [is_sabm(f)][is_disc(f)]
| 1 ! UA ==> continue
2 phl ?[f,f]frame [is_sabm(f)][not(is_disc(f))]
| 1 ! UA ==> continue
3 phl ?[f,f]frame [not(is_sabm(f))][is_disc(f)]
| 1 ! UA ==> continue
4 phl ?[f,f]frame [not(is_sabm(f))][not(is_disc(f))]
| 1 phl ?[f,f]frame ==> continue
```

After simplification, the tree becomes:

```
1 phl ?[f,f]frame [is_sabm(f)][not(is_disc(f))]
| 1 ! UA ==> continue
2 phl ?[f,f]frame [not(is_sabm(f))][is_disc(f)]
```

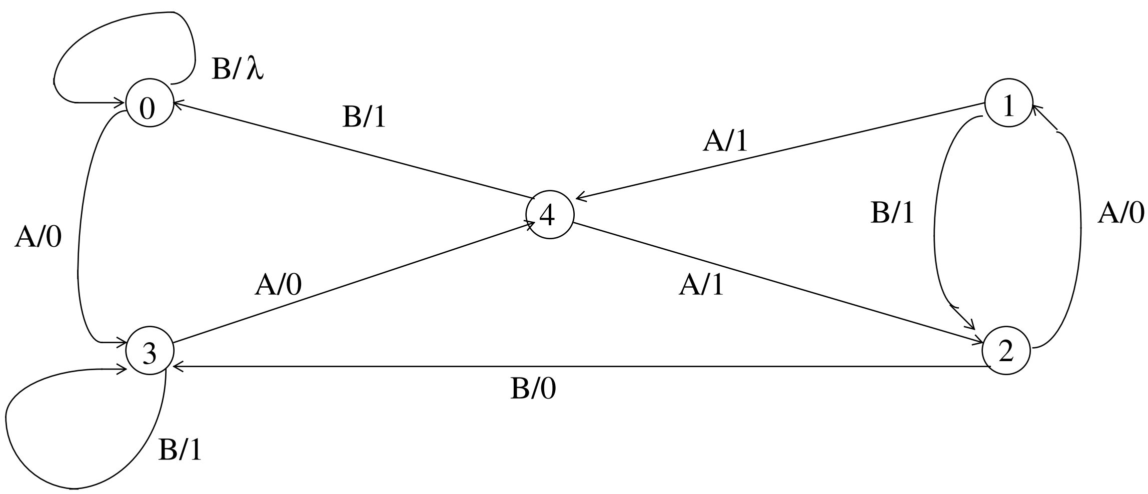
| 1 ! UA ==> continue

3 phl ?[f,f]frame [not(is_sabm(f))][not(is_disc(f))]

| 1 phl ?[f,f]frame ==> continue

4.5.3 The UIO Method

Since the UIO method has been used in the derivation of test sequences from our LAPB LOTOS specification, it is worthwhile to discuss in more details this method and see how it has been adapted to our case. A UIO sequence is a mean for identifying the states of a machine, since it is a unique I/O combination for each state. Assume, for example, the following five-state machine M. This machine has two inputs A and B and two outputs 0 and 1, the symbol λ represents a null output and the representation A/1 means 1 is the output to the input A.



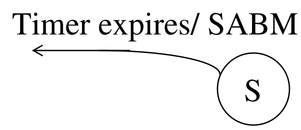
The following table shows a set of UIO sequences for states of M:

State	UIO
0	B/λ
1	A/1 A/1
2	B/0
3	B/1 B/1
4	A/1 A/0

A state can be uniquely identified by observing the output string produced by the application of the input string from its UIO sequence. Thus if the input string is AA and the output is 11, we know that we were at state 1 before the application of AA.

This methodology has been adapted to our purposes. For the tester, the identification of a given state consists of transmitting the input string and then receiving the output string, if the applied UIO sequence consists of an input string and an output string.

Assume that in a given state S a timer is used and if this timer expires an output (SABM) is transmitted; this can be represented by the following figure



This represents the fact that, in state S, if nothing is received and a timeout occurs the output SABM is sent. If the combination timer expires/SABM is unique to the state S, the identification of S can be done as: the tester starts its timer and does not send anything; before its timer expires, the tester should receive the output SABM. This can be written in TTCN:

(Start Timer)

L ? SABM	pass
L ? Otherwise	fail
?Timeout Timer	fail

In the obtained UIO sequences, the input strings consist of at most one input (if the tester does not send anything the input string is null), whereas the output string consists of only one output. This has the advantage of resulting in test sequences which are not too long.

Also a particularity of the DTE LAPB is that it is a non-deterministic protocol. This led us to the following considerations. For example in the state L1, when receiving SABM the DTE responds either with a UA or with a DISC and in state L2 the DTE when receiving SABM responds with a UA. This can be written in TTCN as follows, where Td is a timer duration:

(1) L ! SABM

L ? UA	pass
L ? DM	pass
L ? Otherwise	fail
?Elapse Td	fail

(2) L ! SABM

L ? UA	pass
L ? Otherwise	fail
?Elapse Td	fail

Since in both states L1 and L2 the DTE can respond to SABM with a UA, we cannot distinguish between states L1 and L2. Therefore these two TTCN subtrees cannot be UIO sequences for, respectively, state L1 and L2. In general, if non-determinism is involved for a sequence starting with a transmission of fs in a given state, this sequence is a UIO if all the responses to fs in this state are not responses of fs in all other states.

4.5.4 Test Sequence Derivation Methodology

The test sequence derivation methodology is based on the use of the SSTs and the UIO method. 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 of 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.

The same idea as the one in the LAPB FSM test sequence derivation methodology is used. A test case consists of three parts:

(1) initialization part (initialization step)

(2) body part (evaluation step)

(3) verification part (verification step)

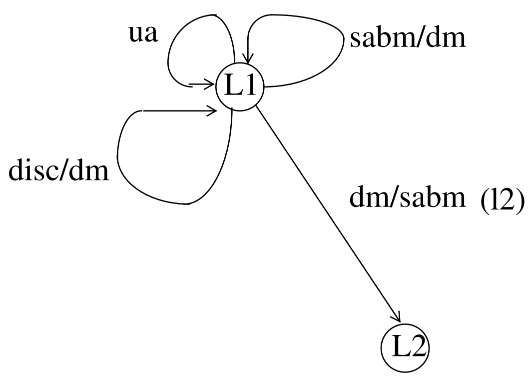
The initialization part consists of bringing the DTE to a given state. The body part consists of performing all the transitions given by the transition tree for that state. The verification part is a unique path which verifies the reached state. Test sequences are translated in TTCN. The translation from the obtained SST to TTCN is straightforward. As already mentioned in 4.1.5, these test sequences use only one PCO, which is called L and corresponds to the gate phl in the specification.

Initialization Step.

The initialization step consists of 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 relation to the 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 Appendix B.

```
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 of 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*, 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

--timeout

+ verification path of what follows

--otherwise: fail

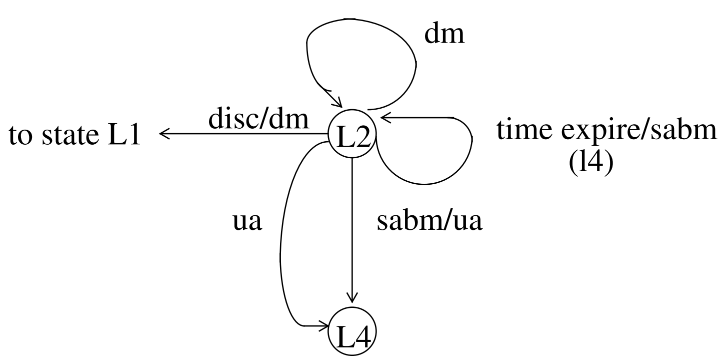
--otherwise fail

-- expiration of timer fail

Consider the previous example with the transition tree (*tt2*) of state *L2* as shown below. This tree can be considered as a small part of the tree called *Linksetup* shown in Appendix B.

```
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 state L1, state L2 and state L4.

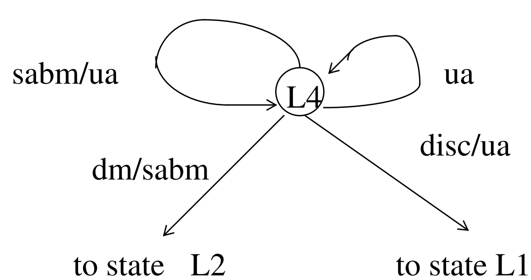
Verification Step.

The verification step consists of finding a verification path. A verification path for a given state is obtained from the transition tree of this state using the UIO method [SD], [SL]. To check that the DTE is at a given state we look for a path 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.

To be a verification path for a given state, a path must not be present in any other transition trees. Consider again the previous example with the transition tree of state L4 as shown in the following. This tree is a small part of the tree called Dataphase shown in Appendix B:

```
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).

In order to be the verification paths of these states, these paths must not be potential verification paths of other states, included those yet to be considered. If (pvp4.1) is the verification path of state L4, a complete test case (2) for state L2 (see Evaluation Step) would be:

+ipl2

L ! SABM

L ? UA

+(pvp4.1)

L ? Otherwise fail

?Elapse Td fail

Resolution of Selection Predicates in SSTs.

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. A selection predicate is associated with this reception. Frame value(s) must be generated from this selection predicate. The frame values which are generated are those for which the selection predicate is true. These frames are the ones to be sent by the tester. To each frame value corresponds a test case.

The selection predicates in the trees shown earlier in the explication of the test case generation methodology are very simple. For example the following frame reception shown in a tree:

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

means that 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, fulfil this selection predicate. Suitable instances of such frames must be generated for testing purposes.

Consider another case in which the selection predicate shown in the tree is defined in the specification as a combination of predicates. The following example illustrates this:

phl ?f : frame[Incorrect(f)]

Assume that 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 the 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. This is done manually.

4.5.5 Discussion of Results

In the following, we discuss the results we have obtained. Starting from the Disconnected State, we give for each phase the initialization path, the transition tree and the verification sequence. Test cases are written in TTCN considering only the dynamic part. In particular, the constraint part is not described. In some trees, actions over the gate dl appear. Gate dl is the gate in which interactions between the DTE LAPB and the network layer occur (figure 3.1). Since the remote test method is used, the interface between the DTE LAPB and the network layer is not accessed. Therefore, the test case generation does not consider actions over gate dl.

The global variables VS and VR are test suite variables used in some test cases. VS and VR denote, respectively, the sequence number of the next I frame to be sent and the sequence number of the next I frame to be received by the tester. The identifiers P, F, NS and NR are field names of the I frames sent or received by the tester (figure 3.2), whereas the identifiers W, X, Y, Z and C/R are field names of an FRMR frame sent or received by the tester (figure 3.3). These fields are declared in the declaration part and the values assigned to them are specified in the constraint part.

The timer TM01 is a tester timer. Its duration Td is specified in the PIXIT. Td indicates the time the tester must wait before determining that the IUT will not respond to a tester stimulus. Timer T1 is a system parameter. Its duration T1 specified in the PIXIT, is used by the IUT. In some trees, some internal events are removed. The trees referred in this section are in Appendix B.

Disconnected State

This state corresponds to state L1 in the LAPB FSM shown in figure 4.6.

Initialization Path: the initialization sequence is given by the TTCN subtree init_disconnect (figure 4.7).

Transition Tree: the Disconnected tree in Appendix B shows such a tree. In this tree, there are six alternatives which correspond to five transitions. Alternative (3) shows an establishment request interaction between the DTE LAPB and the network layer. In alternatives (4) and (5), only the frame values SABM (P:=1) and DM(F:=0), respectively, fulfil the selection predicates associated with these alternatives.

For alternative (4) the following test case is obtained:

+Init_disconnect

L ! SABM (P:=1)

L ? UA [F=1]

+ verif_ua

L ? DM [F=1]

+ verif_disconn

L ? Otherwise fail

?Elapse Td fail

For alternative (5) the following test case is obtained:

+Init_disconn

L ! DM (F:=0)

L ? SABM [P=1]

+ verif_sabm

L ? DISC [P=1]

+ verif_disconn

L ? Otherwise fail

?Elapse Td fail

Verif_disconn is the verification sequence which tests that the DTE is at the Disconnected Phase. Verif_ua is the sequence that tests that the DTE has sent a UA frame in response to a SABM frame (i.e the DTE is at the Data Phase). Verif_sabm is the sequence that verifies that the DTE has sent a SABM frame in response to a DM frame (i.e the DTE is at the Link Set Up Phase).

In alternatives (1), (2) and (6), selection predicates are more complicated. Assume that we denote by f1.1 a frame for which the selection predicate in alternative (1) of the tree is true, by f2.1 a frame for which the selection predicate in alternative (2) of the tree is true and by f6.1 a frame for which the selection predicate in alternative (6) of the tree is true.

Alternative (1) corresponds to the transmission by the tester of a DISC frame or a command frame with the P bit set. A test case for this alternative, where f1.1 is the frame RR(P:=1), could be:

+Init_disconn

L ! RR (P:=1)

L ? DM [F=1]

+ verif_disconn

L ? Otherwise fail

?Elapse Td fail

Alternative (2) corresponds to the transmission by the tester of a frame which is not a SABM, not a DISC and not a DM; such a frame is discarded. A test case for this alternative, where f2.1 is the frame RR(F:=0), could be:

```
+Init_disconn
    L ! RR (F:=0)
      (Start TM01)
         ?Timeout TM01
            + verif_disconn
         L ? Otherwise                    fail
```

Alternative (6) corresponds to the transmission by the tester of an incorrect frame which will be discarded by the DTE. A test case in this category, where f6.1 is the FCS-ERROR, could be:

```
+Init_disconn
    L ! FCS_ERROR
      (Start  TM01)
         ?Timeout  TM01
            + verif_disconn
         L ? Otherwise                    fail
```

Verification Sequence: the transition Disconnected tree characterizes the DTE's Disconnected State. From this tree we obtain the following potential verification paths (for which see the section related to the Link Set Up State):

```
(1)   L ! f1.1
            L ? DM [F=1]          pass
            ?Elapse  Td           fail


(2)   L ! f2.1
            ( Start TM01)
```

?Timeout TM01	pass
L ? Otherwise	fail

(3) L ! SABM (P:=1)

L ? UA [F=1]	pass
L ? DM [F=1]	pass
L ? Otherwise	fail
?Elapse Td	fail

(4) L ! DM (F:=0)

L ? SABM [P=1]	pass
L ? DISC	pass
L ? Otherwise	fail
?Elapse Td	fail

(5) L ! f6.1

(Start TM01)

?Timeout TM01	pass
L ? Otherwise	fail

Link Set Up State

This state corresponds to state L2 in the LAPB FSM shown in figure 4.6.

Initialization Path: this path is obtained by following alternative (5) in the tree Disconnected representing the DTE's Disconnected State. The following initialization TTCN subtree is obtained for the Link Set Up Phase.

+Init_disconn

L ! DM (F:=0)

L ? DISC	inconc
L ? SABM	pass
L ? Otherwise	fail
?ELAPSE Td	fail

Transition Tree: the Linksetup tree in Appendix B shows the transition tree of the Link Set Up DTE's State. This tree shows that after a SABM frame is sent, there are six alternatives which correspond to five transitions. The first alternative correspond to the expiration of the timer. The five other alternatives, which correspond to the five transitions, show the reception by the DTE LAPB of a particular frame.

Alternative (6) corresponds to the transmission by the tester of a frame which is not a DISC, not a SABM, not a UA and not a DM frame with final bit equal to 1. Such a frame is simply ignored. Let us denote by f6.2 such a frame. Test cases obtained for this state are similar to those in [ISO4].

Note that trees tsabm_resolve_collis_0, tsabm_resolve_collis_1 and tsabm_resolve_collis_2 show the DTE's state after a SABM collision in respectively the three cases of collision chosen by the DTE. In the first case the DTE enters the Data Phase only after receiving a UA frame, in the second case the DTE enters the Data Phase after receiving a UA frame or after the timer expires and in the third case the DTE enters directly the Data Phase.

Verification Sequence: from the Linksetup tree, the following potential verification paths are obtained:

```
(1)  (Start  T1)
          L ? SABM [P=1]                        pass
          L ? Otherwise                         fail
          ?Timeout  T1                          fail

(2)      L ! UA (F:=1)
              (Start  TM01)
                  ?Timeout  TM01                pass
                  L ? Otherwise                 fail

(3)   L ! DM (F:=1)
             (Start  TM01)
                ?Timeout  TM01                  pass
                L ? Otherwise                   fail
```

(4) L ! DISC

L ? DM [F=1] pass

?ELAPSE Td fail

(5) L ! SABM (P:=1)

L ? UA [F=1] pass

L ? Otherwise fail

?ELAPSE Td fail

(6) L ! f6.2

(Start TM01)

?Timeout TM01 pass

L ? Otherwise fail

We have to remove the potential verification sequences which are common to the Disconnected State and the Link Set Up State. Verification sequences (3) in Disconnected State and (5) in Link Set Up State are similar. Sequences (2) and (3) in Link Set Up State are included in sequence (2) of Disconnected State, because frame f2.1 can be a UA(F:=1) frame or a DM(F:=1) frame. Sequence (4) in Link Set Up State is included in sequence (1) of Disconnected State, because frame f1.1 can be a DISC frame. Therefore verification sequences (2), (3), (4) and (5) of Link Set Up state are removed from the potential verification sequences of this state.

Sequence (2) of Disconnected State is included in sequence (6) of Link Set Up, because any frame which is an f2.1 frame is a f6.2 frame (selection predicate of alternative (2) in tree Disconnected is included in the selection predicate of alternative (6) of tree Linksetup). Sequence (5) of Disconnected State is included in sequence (6) of Link Set Up, because any frame which is an f6.1 frame is a f6.2 frame (selection predicate of alternative (6) in tree Disconnected is included in the selection predicate of alternative (6) of tree Linksetup). Therefore sequences (2), (3) and (5) of Disconnected State are removed from the potential verification sequences for this state.

Data Phase State

This state corresponds to state L4 in the LAPB FSM shown in figure 4.6.

Initialization Sequence: By following alternative (4) of Disconnected Tree the Data Phase state may be reached. The initialization sequence for this DTE's state:

+tdisconn	
L ! SABM (P:=1)	
L ? DM [F=1]	inconc
L ? UA [F=1]	pass
L ? Otherwise	fail
?Elapse Td	fail

Transition Tree: the Dataphase tree in Appendix B shows such a tree. Test cases obtained for this state are similar to those in [ISO4].

Verification Sequence: the potential verification sequences for this state are derived from the Dataphase tree. According to what we have seen earlier, the potential verification sequences for this state which are present or are included in those of the Disconnected and the Link Set Up States are removed. Only the following potential verification states remain:

(1) L ! RNR (P:=1)	
L ? RR [F=1]	pass
L ? RNR [F=1]	pass
L ? Otherwise	fail
?Elapse Td	fail

(2) L ! RR (P:=1)	
L ? RR [F=1]	pass
L ? RNR [F=1]	pass
L ? Otherwise	fail
?Elapse Td	fail

(3) L ! I (P:=1, NS:=VS,NR:=VR)
(VS:=VS + 1)
L ? RR [F=1] [NR=VS] pass
L ? RNR [F=1] [NR=VS] pass
L ? Otherwise fail
?Elapse Td fail

(4) L ! I (P:=0) (NS:=VS+1)
L ? REJ [F=0] [NR=VS] pass
L ? REJ [P=0] [NR=VS] pass
L ? REJ [P=1] [NR=VS] pass
L ? Otherwise fail
?Elapse Td fail

(5) L ! I (NS:=VS) (NR:=VR+3)
L ? FRMR pass
L ? Otherwise fail
?Elapse Td fail

Note that the potential verification sequences for the Disconnected and the Link Set Up States which are present in the list of potential verification sequences of the Data Phase, are also removed.

Frame Reject State

This state corresponds to state L5 in the LAPB FSM shown in figure 4.6.

Initialization Sequence: by following alternative (7) of Dataphase tree, the tester puts the DTE in the Frame Reject State (i.e the DTE sends an FRMR frame). The initialization sequence for this state is:

+Init_data_phase
L ! I (NS:=VS, NR:= VR+3)
L ? FRMR [WXYZ='0001'B] [C_R=0]

L ? Otherwise fail

?Elapse Td fail

Transition Tree: FRMR tree in Appendix B shows the transition tree for this state. Test cases obtained for this state are similar to those in [ISO4].

Verification Sequence: from the FRMR tree shown in Appendix B, the potential verification sequences for this state are obtained. Those which are present or are included in the potential verification sequence lists of the Disconnected, Link Set Up and Data Phase States are removed. Only the following potential verification sequence remains:

(Start T1)

L ? FRMR [P=1] pass

L ? Otherwise fail

?Timeout T1 fail

Sent Reject State

This state corresponds to state L6 in the LAPB FSM shown in figure 4.6.

Initialization Sequence: by following alternative (27) of tree Dataphase, the tester puts the DTE in Sent Reject State (i.e the DTE sends a REJ frame). The initialization sequence for this state is:

L ! I (NS:=VS + 1,NR:=VR)

L ? REJ [F=0] [NS=VS]

L ? REJ [P=0] [NS=VS]

L ? REJ [P=1] [NS=VS]

L ? Otherwise fail

?Elapse Td fail

Transition Tree: the tree REJ in Appendix B shows the transition tree of this state. Test cases obtained for this state are similar to those in [ISO4].

Verification Sequence: from the transition tree REJ, the potential verification sequences for this state are obtained. Those which are present or are included in the potential verification sequence lists of the Disconnected, Link Set Up, Data Phase and Frame Reject States are removed. Only the following verification sequence remains:

```
(Start T1)
        L ? REJ [P=1]                pass
        L ? Otherwise                fail
        ?Timeout T1                  fail
```

This verification sequence for the Sent Reject State is different from the one in [ISO4]. This is an interesting result (see Section 4.5.6 for further discussion).

Busy State

This state corresponds to state L7 in the LAPB FSM shown in figure 4.6.

Initialization Sequence: by making the tester follow alternative (18) of tree Dataphase, the DTE may enter the Busy State. The initialization sequence for this state is:

```
+init_data_phase
      L ! I (P:=0) (NS:=VS) (NR:=VR)
          L ? RNR
          L ? I                      inconc
          L ? RR                     inconc
          L ? Otherwise              fail
          ?Elapse  Td                fail
```

Transition Tree: the transition tree RNR shows the transition tree for this state. Test cases obtained for this state are similar to those in [ISO4].

Verification Sequence: from the tree RNR the potential verification sequences for this state are obtained. Those which are present or are included in the potential verification sequence lists of the Disconnected, Link Set Up, Data Phase, Frame Reject, Sent Reject and Busy States are removed. Only the following verification sequence remains:

```
L ! I (P=1)(NS:=VS+1)(NR:=VR)
```

L ? RNR [F=1] [NR=VR] pass

L ? DISC inconc

L ? Otherwise fail

?Elapse Td fail

This verification sequence is different from the one in [ISO4] (see 4.5.6).

Send Disc State

This state corresponds to state L3 in the LAPB FSM shown in figure 4.6.

Initialization Sequence: by following alternative (2) of Dataphase tree, the tester may cause the DTE to send a DISC frame. The initialization sequence is:

```
+init_dat_phase
      L ! UA (F:=1)
          L ? DISC [P=1]
          L ? SABM [P=1]                  inconc
          L ? DM [F=0]                    inconc
          L ? Otherwise                   fail
```

Transition Tree: the LINKDISC tree in Appendix B shows the transition tree of this state. Test cases obtained for this state are similar to those in [ISO4].

Verification Sequence: the obtained verification TTCN subtree is similar to the one in [ISO4].

4.5.6 Summary of Main Results

Three main interesting results have been obtained: these are the verification sequences for the Data Phase, Sent Reject and Busy States, which are different from the ones in [ISO4].

The obtained verification sequence for the Data Phase state is:

```
L ! I (P=0, NS:= VS+1, NR:=VR)
    L ? REJ [F=0]                        pass
```

L ? REJ [P=0]	pass
L ? REJ [P=1]	pass
L ? Otherwise	fail
?Elapse Td	fail

This sequence is unique to the Data Phase. The verification sequence for the Data Phase State in [ISO4] is:

L ! RNR (P:=1)

L ? RR [F=1]	pass
L ? RNR [F=1]	pass
L ? Otherwise	fail
?Elapse Td	fail

However, the following TTCN subtree is a potential verification sequence for the Sent Reject State:

L ! RNR (P:=1)

L ? RR [F=1]	pass
L ? REJ [F=1]	pass
L ? Otherwise	fail
?Elapse Td	fail

From these two latter TTCN subtrees it can be seen in both states (Data Phase State and Sent Reject State) that the DTE can decide to respond, to a command RNR (P:=1) frame with a response RR [F=1] frame. Therefore we cannot decide in which state the DTE was (Data Phase State or Sent Reject State) by sending an RNR (P:=1) command frame.

The obtained verification sequence for Sent Reject State is:

(Start T1)

L ? REJ [P=1]	pass
L ? Otherwise	fail
?Timeout T1	fail

This sequence is unique to the Sent Reject State. The verification sequence for the Sent

Reject State in [ISO4] is:

L ! RR (P:=1)
  L ? RR [F=1]   pass
  L ? REJ [F=1]   pass
  L ? Otherwise   fail
  ?Elapse Td   fail

While the following TTCN subtree is a potential verification sequence for the Data Phase State:

L ! RR (P:=1)
  L ? RR [F=1]   pass
  L ? RNR [F=1]   pass
  L ? Otherwise   fail
  ?Elapse Td   fail

As previously, it can be seen from these two latter TTCN subtrees that the DTE can decide to respond to a command RR (P:=1) frame with a response RR [F=1] frame in both Data Phase State and Sent Reject State. Therefore, we cannot decide in which state the DTE was (Data Phase State or Sent Reject State) by sending a command RR (P:=1) frame.

The obtained verification sequence for the Busy State is:

L ! I (P:=1, NS:=VS+1, NR:=VR)
  L ? RNR [F=1] [NR=VR]   pass
  L ? Otherwise   fail
  ?Elapse Td   fail

This sequence is unique to the Busy State. The verification sequence for the Busy State in [ISO4] is:

L ! RNR (P:=1)
  L ? RNR [F=1]   pass
  L ? Otherwise   fail
  ?Elapse Td   fail

While the following TTCN subtree is a potential verification sequence for the Data Phase:

L ! RNR (P:=1)

L ? RR [F=1]	pass
L ? RNR [F=1]	pass
L ? Otherwise	fail
?Elapse Td	fail

From these two latter TTCN subtrees it can be seen that the DTE can decide to respond, during the Data Phase, to a command RNR (P:=1) frame with a response RNR [F=1] frame. Therefore we cannot distinguish between the Data Phase State and the Busy State by sending a command RNR (P:=1).

4.6 Comparison

The earlier work of generation of test cases for a DTE LAPB [Kan] was done on the basis of the transition table of the DTE LAPB. This work was facilitated by the fact that the FSM model for protocol specification is well developed and there are many tools for its verification. Hence, test suites can be correctly developed using the validated FSM specification for the protocol. Another reason for using this method is that the three step method for generating test cases is easy to use with a FSM model and considers the important cases of proper, improper and inopportune test stimuli.

However, in general, the translation of a protocol informal description to a finite state machine poses some difficulties. There is no known algorithm reducing the finite state machine so that the result is minimal, completely specified and is a true translation of the informal description. There is a possibility that the number of states in the FSM model would be very large. In that case, the state transition table will be very large also, and testing all the transitions for proper, improper and inopportune PDUs will become very time expensive. Finally, because it is a semi-formal description, the FSM model does not describe clearly some aspects of a protocol. For example, the timing is not clearly specified. Consequently, one has to refer to the informal specification too, when doing the test case generation, in order to see how the aspects not clearly specified in the FSM are described.

On the other hand, because it is written in the formal description technique LOTOS, our LAPB specification entirely and clearly describes the behaviour of the DTE LAPB. Therefore, the test case generation can be done solely from the LOTOS LAPB specification. Actually, LOTOS describes more clearly and more precisely protocols than state tables, mainly for two reasons. The first reason is that LOTOS formalizes the data, whereas state tables do not. The second reason is that certain aspects of the protocol are not clearly described by state tables, like the timing.

The first point one has to deal with, when writing a protocol specification in LOTOS, is to adopt a specific specification style. This style will have effects on the size of the specification, the ease of simulating and debugging it and on the test case generation step, if generation of test cases is a purpose of writing the specification. In order to generate test cases from a LOTOS specification, one has to find an appropriate methodology. Unfortunately, no methodology is known for generating test cases from a real-life LOTOS specification. We have seen that the theory discussed in Section 4.3 is severely limited in practice. Therefore we used an adhoc method, combining the use of symbolic trees with the UIO method. Our methodology may not be practical for much larger specifications, because of the use of symbolic trees.

The use of the UIO method has led us to find three interesting results. These are the verification sequences for the Data Phase State, the Sent Reject State and the Busy State. The sequences found allow to uniquely identify these states.

Chapter 5. Conclusion and Suggestions for Future Work

5.1 Summary of the Thesis

We have developed a LOTOS specification for LAPB, the X.25 data link layer, and we have described the derivation of test cases from this specification by using the LOTOS interpreter ISLA. First, we started by introducing some concepts related to the FDT LOTOS. We have, then, described the design of the LAPB specification in LOTOS. We have shown that, in general, LOTOS specifications can be written in different styles. Because our specification was intended to be used for test cases, we have seen why the most effective design style in LOTOS, the constraint-oriented style, was not used.

After briefly describing FSM test case derivation methodologies, we presented the methodology used in generating test cases for the LAPB FSM. We have shown that, in the case of LOTOS, the only test case derivation methodology that exists applies to a very limited subset of the language only. Therefore, it was not possible to use this methodology in our case. We presented the methodology of generating test cases from our LAPB LOTOS specification, which combines the use of the symbolic trees generated by the interpreter with the use of the UIO method.

Some features were added to the LOTOS interpreter to considerably reduce the size of the generated trees (Section 4.5.2). Finally, we have seen that some additional interesting results were found in our methodology, in comparison with those obtained from the LAPB FSM.

5.2 Suggestions for Future Work

The LAPB protocol has been completely specified in LOTOS. A possible next step would be to specify in LOTOS the optional Multi-Link Procedure (MLP), by using the existing LAPB specification. The MLP is used for data interchange over one or more single link procedure. Optional test cases can, then, be generated from the LOTOS specification. These test cases would have to be passed when support of MLP is claimed. Note that test cases for MLP are not specified in [ISO4].

Another more challenging task would be to specify the packet level of X.25 in LOTOS. It would be interesting to try to apply our methodology to this level, to see if it is still practical.

A software tool added to the LOTOS interpreter which allows to check if a trace, written as LOTOS process, is part of a LOTOS specification, is being implemented at University of Montreal. This check is done by synchronizing the trace with the specification. If the trace is not part of the specification, some diagnostics are given by the tool. This tool could be used in our case after translating the given test cases written in TTCN into LOTOS processes.

Furthermore, additional research in this area may show how the method used here could be made more powerful and possibly automated to some extent with the use of appropriate tools, in order to make our methodology practical on a larger scale.

Derivation of test cases from LOTOS specifications is a very new topic, which is still in the preliminary stage of research. To our knowledge, this is the first work that demonstrates test case generation from a real, large protocol specification written in LOTOS.