# A NEW METHODOLOGY FOR OSI CONFORMANCE TESTING BASED ON TRACE ANALYSIS

By

RUSSIL WVONG

B. Sc., The University of British Columbia, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming

to the required standard

....................................

....................................

THE UNIVERSITY OF BRITISH COLUMBIA

October 1990

# Abstract

This thesis discusses the problems of the conventional ISO 9646 methodology for OSI conformance testing, and proposes a new methodology based on *trace analysis*. In the proposed methodology, a *trace analyzer* is used to determine whether the observed behavior of the implementation under test is valid or invalid. This simplifies test cases dramatically, since they now need only specify the expected behavior of the IUT; unexpected behavior is checked by the trace analyzer. Test suites become correspondingly smaller. Because of this reduction in size and complexity, errors in test suites can be found and corrected far more easily. As a result, the reliability and the usefulness of the conformance testing process are greatly enhanced.

In order to apply the proposed methodology, trace analyzers are needed. Existing trace analyzers are examined, and found to be unsuitable for OSI conformance testing. A family of new trace analysis algorithms is presented and proved.

To verify the feasibility of the proposed methodology, and to demonstrate its benefits, it is applied to a particular protocol, the LAPB protocol specified by ISO 7776. The design and implementation of a trace analyzer for LAPB are described. The conventional ISO 8882-2 test suite for LAPB, when rewritten to specify only the expected behavior of the IUT, is found to be more than an order of magnitude smaller.

# Contents

# List of Figures

# Acknowledgements

I would like to thank my supervisor, Dr. Gerald Neufeld, for his advice and encouragement throughout the writing of this thesis. I would also like to thank Dr. Samuel Chanson, for reading through the first draft; the faculty, staff, and grad students of the Department of Computer Science, for providing a friendly and supportive environment; and the Natural Sciences and Engineering Research Council of Canada, the B. C. Advanced Systems Institute, and IDACOM Electronics Ltd., for their financial support. Finally, I would like to thank my family, and especially my parents, for their support and guidance.

# Chapter 1

# Introduction

This thesis discusses the problems of the conventional ISO 9646 methodology for OSI conformance testing, and proposes a new methodology based on *trace analysis*. In the proposed methodology, a *trace analyzer* is used to determine whether the observed behavior of the implementation under test is valid or invalid. This simplifies test cases dramatically, since they now need only specify the expected behavior of the IUT; unexpected behavior is checked by the trace analyzer. Test suites become correspondingly smaller. Because of this reduction in size and complexity, errors in test suites can be found and corrected far more easily. As a result, the reliability and the usefulness of the conformance testing process are greatly enhanced.

This chapter provides the motivation for the new methodology. It describes the problems of the conventional methodology, and shows how using a trace analyzer overcomes those problems. Later chapters discuss specific trace analysis methods, and demonstrate the feasibility of the new methodology by applying it to the LAPB protocol.

## 1.1 The conventional methodology

We begin by discussing why OSI conformance testing is needed. We then describe the conventional ISO 9646 conformance testing methodology and discuss the problems with it.

### 1.1.1 Conformance testing

ISO is currently standardizing computer communication protocols within the framework of the OSI reference model. The idea is that any two systems which implement the same OSI protocols will be able to communicate with one another. For the reader who is unfamiliar with communication protocols and the OSI protocol standards, Tanenbaum [18] provides a good introduction.

Unfortunately, the standards documents which define the protocols are often unclear or ambiguous. They may be interpreted by different people in different ways. As a result, different implementations of the same protocol sometimes turn out to be incompatible with one another.

To deal with this problem, ISO is also standardizing certain *conformance testing* procedures, which are carried out on a given protocol implementation to ensure that it conforms to the appropriate protocol standard. The idea is that these tests will be performed by some recognized test laboratory. After a protocol implementation passes the tests, the laboratory gives it a certification of approval. Customers may then buy the protocol implementation from its vendor with at least some confidence that it will be compatible with other implementations.

The client for whom the laboratory performs the conformance tests may be either the implementation vendor, wanting to use the certification of conformance as a selling point; or a potential customer, wanting some assurance from a third party that the implementation works. In either case, the test laboratory has little or no knowledge of the internal workings of the system under test (SUT) which contains the implementation (IUT). The conformance testing process treats the SUT as a black box: only its external behavior is observed.

### 1.1.2 The ISO 9646 methodology

The conventional conformance testing methodology is described by the five-part standard ISO 9646. In this methodology, the IUT is not simply run against another implementation, as this would not test the reaction of the IUT to invalid messages. Instead, a *tester* able to send both valid and invalid messages is used to execute a series of *test cases*, each testing a particular aspect of the protocol. For example, there is usually one test case for each possible state transition.

There are four different test architectures discussed in ISO 9646, called

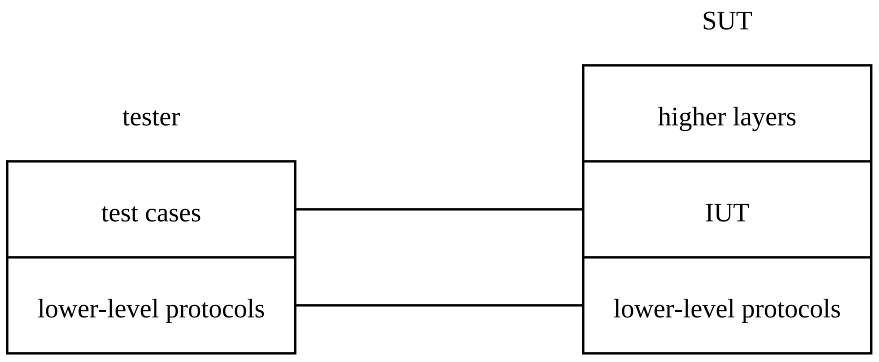

Figure 1.1: The basic conformance testing architecture

"methods": local, coordinated, distributed, and remote. The local test method is not intended for conformance testing, since it requires access to the lower service boundary of the IUT, which is internal to the SUT; such access is not available during third-party testing. In the other three test methods, the tester exchanges messages with the IUT over a normal communications service, provided by lower-level protocol implementations in both the tester and the SUT (Figure 1.1).

In the remote test method, the tester interacts with the SUT only via the messages which it sends and receives. In the coordinated and distributed test methods, the tester also has a point of control and observation at the upper service boundary of the IUT. The remote test method is generally considered the most practical architecture, since it is the only one which does not require access to any internal interfaces within the SUT.

ISO is planning to standardize an *abstract test suite*, consisting of *ab-*

*stract test cases*, for each of the OSI protocols. The abstract test suite for each protocol specifies all the tests which an implementation must pass in order to receive its certification of conformance. TTCN, the Tree and Tabular Combined Notation, is used to specify abstract test suites. In this notation, each test case is specified as a tree of events (such as: tester sends message to IUT, tester receives message from IUT, timer expires). Each path from the root of the tree to one of its leaves is a possible sequence of these events. Each leaf has a verdict associated with it: PASS, if the sequence of events corresponds to the expected IUT behavior; FAIL, if the sequence of events corresponds to invalid IUT behavior; or INCONCLUSIVE, if the IUT's behavior was valid, but not what was expected. TTCN has many of the features of a programming language—variables, expressions, loops, subroutines, and so on—but being a special-purpose notation, is somewhat limited when compared to a general-purpose language like C or Pascal.

The abstract test cases are implemented as *executable test cases* on particular testers. TTCN is precise enough and explicit enough to permit direct translation from abstract test cases to executable test cases: in fact, at least two systems which are able to perform such translation automatically have been built [20, 16]. Thus TTCN can be viewed as a high-level programming language: once a TTCN test suite has been written, it can be translated directly into executable code without human intervention.

### 1.1.3 Problems with the test suites

At present, two TTCN test suites have been specified by ISO, one for the X.25 LAPB protocol and one for the X.25 packet layer protocol. These test suites are specified by Parts 2 and 3, respectively, of the document ISO 8882 [7]. They are being standardized separately.

Unfortunately, both test suites have a number of major problems. They contain many, many errors, despite repeated revisions by protocol experts over a period of years. (Errors in the current version of ISO 8882-2 are described in Chapter 4.) The presence of such errors means that we can only have a limited amount of confidence in the results of conformance testing.

Moreover, the test suite standards themselves are incredibly large, much larger than the original protocol standards from which they are derived. For example, the ISO protocol standard for LAPB, ISO 7776, is only 23 pages long; the current version of ISO 8882-2 is 252 pages long, an entire order of magnitude larger. Documents of this size are very difficult to read and understand; again, this lowers our confidence in the results of conformance testing.

In short, the existing abstract test suites are *untrustworthy.* This calls into question the usefulness of conformance testing. If an IUT fails certain conformance tests, is it because of errors in the IUT, or because of errors in the test cases? In order to establish the usefulness of conformance testing, some way of producing reliable, trustworthy test suites must be found.

### 1.1.4 Problems with the methodology

Why do existing test suites have the problems which we have described? The reason appears to lie in the conformance testing methodology, rather than the particular protocols being tested. Each TTCN test case must specify all possible valid sequences of events, both expected and unexpected. It is surprisingly difficult to identify all unexpected but valid sequences of events, for a number of reasons. The most important reason is that *the order in which events are observed by the tester may not be the same as the order in which they are observed by the IUT.*

It may not be immediately obvious why this is so. To see why, consider the common situation in which a test case sends a message $q$ to the IUT, in order to verify that the IUT sends a response $r$. The expected sequence of events in this case is $(q, r)$.

The message $q$ takes some amount of time to travel from the tester to the IUT. Suppose that during this time, the IUT sends a message of its own, call it $p$. Then the tester sees the sequence of messages $(q, p)$—it receives the message $p$ after it sends $q$—whereas the IUT sees the sequence $(p, q)$ (Figure 1.2).

According to the protocol, the IUT is required to send $r$ immediately upon receiving $q$; it is invalid for the IUT to send $p$ after receiving $q$. Nevertheless, the test case must specify both $(q, r)$ *and* $(q, p)$ as being valid sequences of events—the former being the expected behavior, and the latter being unexpected but valid behavior—because the sequence $(q, p)$ observed

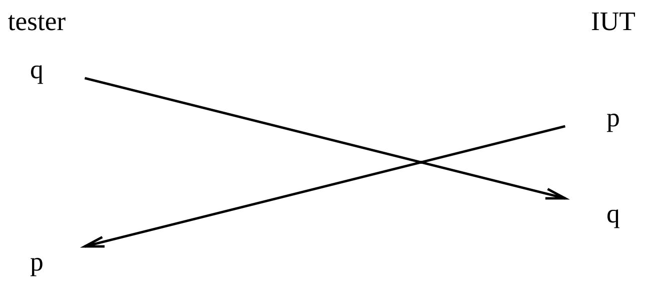

Figure 1.2: Different sequences of events at the tester and at the IUT

by the tester may have been caused by the sequence $(p, q)$ at the IUT, which is valid.

It should be clear from this example that identifying all possible valid-but-unexpected sequences of events is not easy. As a result, individual test cases are rather more complex than might be expected. There is also considerable redundancy among the test cases, since TTCN has somewhat limited facilities for identifying functions common to different test cases. A test suite consisting of hundreds of these test cases will therefore be very large. As described earlier, TTCN is basically a programming language, so a TTCN test suite is essentially a very large and complex program. Writing and "debugging" a TTCN test suite is not an easy task, especially since there are few support tools for TTCN. Frequent errors are inevitable.

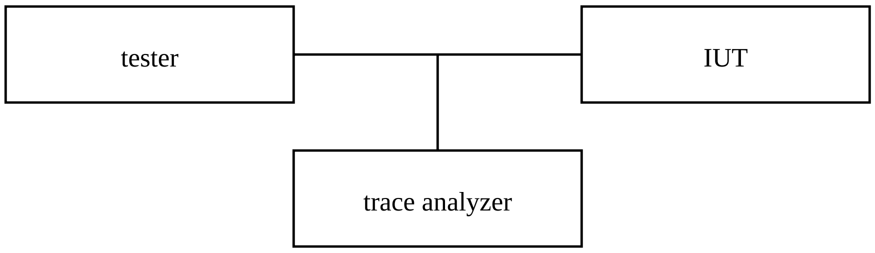

Figure 1.3: Using a trace analyzer for conformance testing

## 1.2 A methodology based on trace analysis

This thesis proposes an alternative methodology for conformance testing. The basic idea is that instead of having each test case identify all unexpected but valid sequences of events, a *trace analyzer* is used to determine whether or not the observed behavior of the IUT is valid. The individual test cases need only specify the *expected* sequence of events; any unexpected events will be checked for validity by the trace analyzer. The trace analyzer passively observes the exchange of messages between the tester and the IUT (the *trace*), notifying the test operator whenever it discovers an error in the IUT's behavior (Figure 1.3). No new specification need be given for the trace analyzer: it can be implemented directly from the protocol specification.

In the new methodology, test case verdicts may be assigned in a straightforward manner. If the IUT's behavior is found to be invalid by the trace analyzer, the verdict is FAIL. If the IUT's behavior is valid, and matches what was expected by the test case, the verdict is PASS. If the IUT's behavior is valid, but unexpected, the verdict is INCONCLUSIVE.

The new methodology dramatically simplifies test cases, since they only specify the expected sequence of events. The size of the test suite is correspondingly reduced; for example, the ISO 8882-2 test suite is reduced from 252 pages to 8 pages. (The re-specified test suite is described in Chapter 4.) This reduction in size and complexity makes it far easier to detect and correct errors in test cases.

An additional advantage is that the problem of determining the validity of the behavior of the IUT is handled in one place—the trace analyzer—instead of being distributed over hundreds of test cases. This eliminates a considerable amount of redundancy, and makes the handling of this problem easy to change.

In short, the proposed methodology should make test suites trustworthy, and greatly enhance the reliability and usefulness of the conformance testing process.

## 1.3 Other possible solutions

One might ask whether it is *necessary* to introduce trace analysis in order to make conformance testing useful. For example, why not simply continue debugging the abstract test suites until no errors remain? Wouldn't this be easier than adopting a new methodology?

It may be possible to do this, but it doesn't appear very likely. As discussed earlier, the abstract test suites are essentially large and complex programs, written in a somewhat unstable language (TTCN) with which we have little experience and for which there are few support tools. It is unreasonable to expect that the test suites can be debugged in the near future. Indeed, the ISO 8882-2 and ISO 8882-3 test suites have been revised periodically for a number of years, but still contain many errors.

A second criticism that might be made is that errors in the abstract test suites are not important. Won't errors be discovered as the test cases are implemented? And even if they are not, can't the test laboratory personnel check the test case verdicts by hand?

It is true that if the abstract test cases are being implemented manually, some of the errors in them will be detected and corrected. However, if all of ISO's protocol experts are unable to detect the errors in an abstract test suite, it seems improbable that a test suite implementor will.

With regard to test laboratory personnel checking the test verdicts—it is possible to do this, provided that the people involved are experts in the protocol being tested, and are willing to go through the tedious task of wading through the several hundred pages of output generated by the test cases. However, this approach is undesirable for two reasons. First, it changes the conformance testing process from a more or less automated one—the test cases can be executed against the IUT with little or no human intervention—to one requiring considerable human expertise. Second, the test laboratory personnel would basically be serving as human substitutes for a trace analyzer. Considering the repetitive and mechanical nature of trace analysis (various algorithms are discussed in Chapter 2), this seems like a waste of human effort.

A third critic might ask whether the problem might not be solved simply by abandoning TTCN and the idea of automatic implementation of abstract test cases. Suppose we re-specify the test cases to identify only the expected sequence of events, so that the abstract test suites become simpler and smaller, as described earlier. Then what do we need a trace analyzer for?

The answer is that this approach merely shifts the problem of specifying unexpected but valid sequences of events from the abstract test cases to the executable test cases. It is now the test suite implementor who is responsible for identifying all unexpected but valid sequences of events. The same comments about the difficulty of identifying unexpected but valid sequences of events, the complexity of the resulting test cases, the size of the resulting test suite, and the difficulty of debugging it apply just as well to the executable test cases as to the abstract test cases.

Finally, in recent years some work has been done on automatic generation of test suites from formal protocol specifications. Wouldn't this solve the problem? Wouldn't test suites generated by machine would be free of errors?

Perhaps at some time in the future, this may be a viable strategy. At present, though, there are some major obstacles. Present methods for test case generation, surveyed by Sidhu and Leung [17], only generate the expected sequence of events; the problem of unexpected but valid events is not addressed at all. Moreover, protocol standards are currently written in English; no formal specifications have been standardized yet. Even if they were, there is presently a large gap between the formal specification techniques used to specify protocols, such as Estelle and LOTOS, and the state machine specifications used to generate test suites. This problem is discussed by Miller [12].

## 1.4 Overview of the thesis

In this chapter, we have discussed *why* trace analysis should be used to do conformance testing. In Chapter 2, we discuss *how* it should be done: the requirements for a trace analyzer used in conformance testing are discussed, existing trace analyzers and trace analysis methods are examined and found wanting, and a family of new trace analysis algorithms is presented and proved.

Chapters 3 and 4 demonstrate the feasibility of the proposed methodology, by applying it to the LAPB protocol. Chapter 3 discusses the design and implementation of a trace analyzer for the LAPB protocol; Chapter 4 discusses the effects of re-specifying the ISO 8882-2 test suite according to the new methodology.

Finally, Chapter 5 presents the conclusions of the thesis.

## 1.5 Related work

The idea of automated trace analysis is not new. Trace analyzers for various protocols are described by Cork [3]; Molva, Diaz, and Ayache [13]; Matthews, Muralidhar, and Sparks [11]; Probert [15]; and Lo [10]. More general approaches to trace analysis are discussed by Ural and Probert [19]; Bochmann, Dssouli, and Zhao [2]; and Bochmann and Bellal [1]. However, all of these trace analyzers and trace analysis methods are unsuitable for *conformance* testing (as opposed to, say, diagnostic testing); the reasons are discussed in Chapter 2.

With respect to methodology, Bochmann, Dssouli, and Zhao [2] also discuss the idea of separating trace analysis from individual test cases. They suggest that a trace analyzer would be useful for validating TTCN test cases, or in situations in which standard test cases cannot be applied, but stop short of suggesting, as we have, that a conformance testing methodology based on trace analysis could be used to replace the ISO 9646 methodology altogether.

The conventional conformance testing methodology is described by the five-part standard ISO 9646 [8]. The three-part standard ISO 8882 [7] specifies the standard TTCN test suites for the X.25 LAPB and packet layer protocols; the development of the LAPB test suite is described by Kanungo, Lamont, Probert, and Ural [9].

Chapter 2

# Trace analysis methods

We now turn to the question of *how* trace analysis should be done. This chapter discusses the requirements for a trace analyzer used in conformance testing, and examines existing trace analysis methods in light of these requirements. Finally, three new trace analysis algorithms are presented and proved.

## 2.1 Requirements

To apply the conformance testing methodology proposed in Chapter 1, a trace analyzer is needed. However, not just *any* trace analyzer is suitable; to be useful in conformance testing, it should satisfy the following requirements.

First, the trace analyzer should require minimal human intervention. A typical test suite contains hundreds of test cases, takes hours to run, and generates hundreds of pages of traces. Clearly, the trace analysis process should be as automated as possible. Requiring that a protocol expert double-check each test case verdict is not acceptable.

Second, the trace analyzer should assign a FAIL verdict only if it is *certain* that an error has occurred. This is a consequence of the first requirement: If the trace analyzer were to flag any event that looked as though it *could* be caused by an IUT error, human intervention would be required to decide whether an error had in fact occurred.

Third, the trace analyzer should treat the SUT as a black box. It is unreasonable to expect the implementation vendor to expose interfaces within the SUT in order to do conformance testing.

Fourth, the trace analyzer should not assume that the order in which events are observed is the same as the order in which they occur at the IUT; because of propagation delay between the observer and the IUT, the order may be different, as discussed in Chapter 1. Also, if the underlying communications service is unreliable, the trace analyzer should take this into account.

The discussion above refers to a trace analyzer for a specific protocol, but more generally, any trace analysis algorithm should also satisfy these requirements. In addition, to be most useful, a trace analysis algorithm should make minimal assumptions about the protocol being tested. For example, a trace analysis algorithm would be of limited usefulness if it assumed that the protocol being tested is deterministic, since most protocols are not.

A final consideration is whether the trace analyzer or trace analysis algorithm is able to operate on-line, as the test cases are being executed, or must analyze the recorded traces off-line. The former would be more useful, since it can detect errors as they occur. Note that any trace analysis algorithm which can run on-line can also run off-line, but not vice versa.

## 2.2 Existing trace analysis methods

When existing trace analyzers are examined, it is found that there are only two basic methods used: heuristics and simulation. Trace analyzers which use heuristics [11, 15] check the given trace against some predefined set of rules; anything which looks like an error is marked. Thus a heuristic trace analyzer relies on a human protocol expert to check its results. As discussed above, a trace analyzer which requires extensive human intervention is not suitable for conformance testing (although it may be useful for other applications).

Other trace analyzers [3, 13, 1, 10] simulate a perfect implementation, and compare its behavior to the behavior of the IUT. All messages received by the IUT are sent to the simulated implementation, and the messages sent by the IUT are compared to the messages sent by the simulation. Most of the trace analyzers assume that the protocol is deterministic. [19, 1] describe trace analysis algorithms which do not make this assumption, but these algorithms use backtracking to handle non-determinism, which prevents them from being used on-line.

All existing trace analyzers and trace analysis methods which use simulation have this in common: they assume that events are observed in the same order as they occur at the IUT. As discussed in Chapter 1, this is not a valid assumption.

In summary, existing trace analysis methods are unsuitable for OSI conformance testing. New methods are needed.

## 2.3 A family of new trace analysis algorithms

We now present and prove three new trace analysis algorithms. They can be used when testing any protocol which can be specified as an extended finite state machine; the protocol need not be deterministic. The algorithms operate by processing all observed messages in order, keeping track of the possible states of the IUT, rather than using backtracking; hence they can be run either on-line or off-line.

The individual algorithms differ in their assumptions about the trace. The first algorithm, which is presented solely for purposes of illustration and is not meant to be used in practice, requires that messages actually be observed within the SUT, at the lower service boundary of the IUT. Since there is no propagation delay between the IUT and the observer in this case, the algorithm can assume that all messages are observed in the same order as they are sent and received by the IUT. With this assumption, the algorithm is fairly trivial.

The second algorithm places the observer outside the IUT. In this situation, messages may not be observed in the same order as they are sent and received by the IUT; the algorithm must take this possibility into account. However, the second algorithm does make the assumption that the underlying communications service is reliable, and will not lose messages or deliver them out of sequence.

Finally, the third algorithm assumes that messages may be lost by the underlying communications service.

In general, as the assumptions are made weaker, the algorithm becomes more complex, and its error-detecting power decreases: it becomes more difficult to be *certain* that an error has occurred.

### 2.3.1 Model of the IUT

The IUT is modelled as an extended finite state machine. At any given moment, the IUT has an identifiable state, which determines its possible reactions to subsequent events. The IUT may change to a different state when it receives a message or when an internal event occurs, such as a timeout or a request from a higher layer, possibly sending one or more messages of its own. In some cases, there may be more than one possible state transition which the IUT can make; one of them will be chosen when the event occurs. State transitions are atomic: only one occurs at a time. A state machine may be represented as a directed graph, with the nodes of the graph representing states and the arcs representing state transitions; the

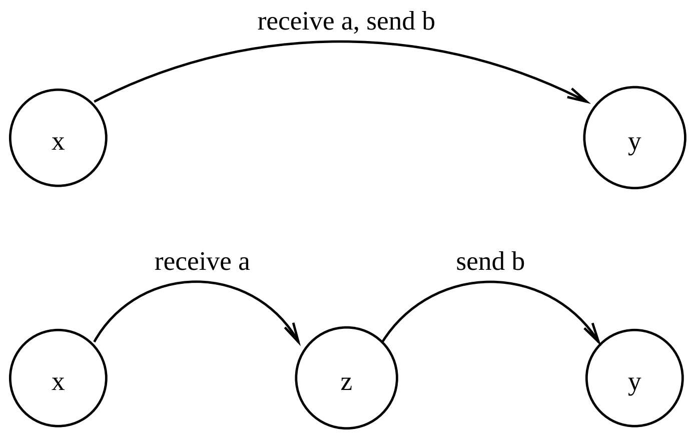

Figure 2.1: Decomposing a state transition

arcs are labelled with messages received and sent during the state transition.

The trace analyzer can only observe one message at a time. Therefore, it is awkward to handle a state transition involving more than one message. However, we can assume without loss of generality that each state transition has at most one message associated with it, either sent or received: If a single state transition has more than one message associated with it, we can decompose it into several transitions, each having a single message associated with it, with corresponding intermediate states. For example, suppose that the IUT may change from state $x$ to state $y$ when it receives a message $a$, sending message $b$ in response. We can decompose this transition into two transitions, as shown in Figure 2.1.

There may be state transitions which have *no* messages associated with them. That is, the IUT may change state without having sent or received a message. For example, such transitions might be caused by internal events (such as timeouts or user requests). We will refer to such transitions as *invisible transitions.* They are not visible to the trace analyzer, which only observes messages.

Let $Q$ be the set of all possible states of the IUT. Let $A$ be the set of messages which can be sent or received by the IUT; a message which can be both sent and received by the IUT is represented by two different elements of $A$. If $q \epsilon Q$ is a state of the IUT, then for each $a \epsilon A$ representing a message which may be received by the IUT, $a(q)$ denotes the set of possible states of the IUT after it receives $a$ in state $q$; and for each $a \epsilon A$ representing a message which may be sent by the IUT, $a(q)$ denotes the set of possible states of the IUT after it sends $a$ in state $q$. $a(q)$ typically contains a single state, but may be empty (because event $a$ cannot occur in state $q$) or contain more than one state (because of nondeterminism). If $S \subseteq Q$ is a set of states, $a(S)$ will denote the set $\bigcup_{q \epsilon S} a(q)$, that is, the set of all states which may be reached from a state in $S$ via event $a$.

In addition, we define $c(S)$, the *completion* of a set $S$ with respect to invisible transitions, to be the set of all states reachable from some state in $S$ via zero or more invisible transitions. In particular, $c(S)$ includes $S$. Whenever it is possible for the IUT to be in some state in $S$, it is also possible for it to be in a state in $c(S)$, since the IUT may change from a state in $S$ to a state in $c(S)$ without any messages being observed.

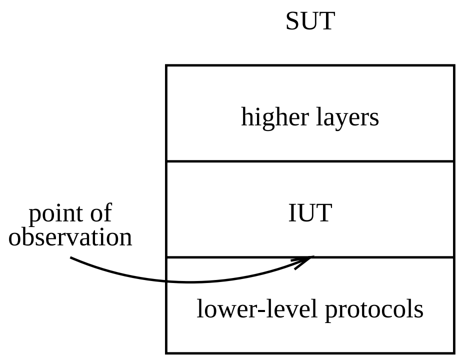

Figure 2.2: An observer within the SUT

### 2.3.2 Algorithm 1: observer within the SUT

We begin with a trivial algorithm, assuming that the lower service boundary of the IUT can be observed directly (Figure 2.2). Under this assumption, all messages are observed as they are sent and received by the IUT, with no intervening delay. There is no difference between the order in which messages are observed by the trace analyzer and the order in which they are sent and received by the IUT. Needless to say, this assumption is completely unrealistic for conformance testing; the algorithm is presented only for illustrative purposes.

Let $a_1, a_2, \ldots$ be the sequence of messages observed. For $i = 0, 1, 2, \ldots$, define $S_i$ to be the set of possible states of the IUT after it has sent or received the messages $a_1, a_2, \ldots a_i$. Initially, the IUT can be in any state, so $S_0 = Q$.

*Lemma 1.* $S_i = c(a_i(S_{i-1}))$.

*Proof.* By definition.□

*Algorithm.* Let $s := Q$.

When message $a_i$ is observed, set $s := c(a_i(s))$.

If $s$ is now empty, the IUT fails.

Otherwise, wait for the next message and repeat.

*Invariant.* At any given time, $s = S_i$, where $i$ is the index of the last message observed.

*Proof of invariant.* By induction.

Initially, $s = S_0$; the invariant is trivially true.

Now suppose that $a_{i-1}$ was the last message observed, and that $s = S_{i-1}$, by inductive hypothesis. When $a_i$ is observed, $s$ is assigned the value $c(a_i(s)) = c(a_i(S_{i-1})) = S_i$, so the invariant is preserved.□

It follows that if $s$ ever becomes empty, then $S_i$ must be empty for some $i$: that is, after the IUT sends or receives the message $a_i$, there is no valid state which it could be in. Therefore, the IUT must be incorrect.

Essentially, this algorithm simulates a nondeterministic state machine using a deterministic one. See Theorem 2.1 of Hopcroft and Ullman [4].

### 2.3.3 Algorithm 2: observer outside the SUT

The first algorithm assumes that the observer is placed inside the SUT, which is completely unrealistic. A more realistic setup is one in which the observer is placed outside the SUT (Figure 2.3). In this situation, however, the order in which messages are observed may not be the same as the order

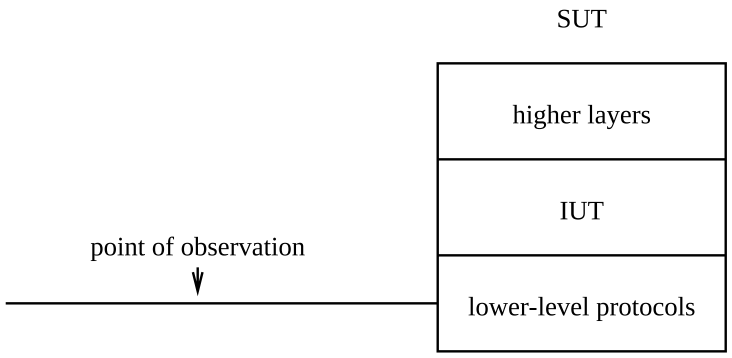

Figure 2.3: An observer outside the SUT

in which they are sent and received by the IUT, as discussed in Chapter 1.

**Partial ordering of messages**

We define a partial ordering $\prec$ on the messages observed by the trace analyzer: if we know for certain that message $m$ must have been sent or received by the IUT before message $m'$, based only on the information available to the trace analyzer, then we write $m \prec m'$.

Given any two messages $m$ and $m'$, there are three cases: $m \prec m'$; or $m' \prec m$; or *neither*, that is, it is possible that $m$ was sent or received before $m'$, but it is *also* possible that $m'$ was sent or received before $m$. In the third case, we write $m \nprec m'$ and $m' \nprec m$.

How can we tell which case applies? Suppose that a timestamp is assigned to each message observed. Let $t(m)$ be the time at which message $m$ is observed.

Suppose that $p$ and $p'$ are messages sent by the IUT. If $t(p) < t(p')$, then we know that the IUT must have sent $p$ before it sent $p'$, because the underlying communications service delivers messages in order (by assumption); so $p \prec p'$. Similarly, if $q$ and $q'$ are messages sent to the IUT, such that $t(q) < t(q')$, then the IUT must receive $q$ before it receives $q'$, so $q \prec q'$.

Suppose that $p$ is sent by the IUT and $q$ is sent to the IUT. There are three possible cases. We know that if $t(p) < t(q)$, then the IUT must have sent $p$ before it received $q$, because at the time $p$ is observed, the IUT cannot have received $q$ yet; so $p \prec q$.

Let $\delta$ be the maximum round-trip propagation delay between the observer and the IUT: that is, the maximum time it takes for a message to travel from the observer to the IUT, for the IUT to send a response, and for the response to travel from the IUT to the observer. If $t(p) > t(q) + \delta$, then the IUT must have sent $p$ after it received $q$, because if not, the round-trip propagation delay would have been greater than $\delta$, which is impossible (Figure 2.4). So $q \prec p$.

In the final case, $t(q) < t(p) < t(q) + \delta$. In this case, $p \not\prec q$ and $q \not\prec p$: that is, we cannot tell whether $q$ was received before $p$ was sent, or vice versa, even though $q$ was observed before $p$.

Lemma 2 summarizes what we know about the partial ordering $\prec$.

*Lemma 2.* Let $p$, $p'$ be messages sent by the IUT, and let $q$, $q'$ be messages sent to the IUT.

(i) If $t(p) < t(p')$, then $p \prec p'$. If $t(q) < t(q')$, then $q \prec q'$.

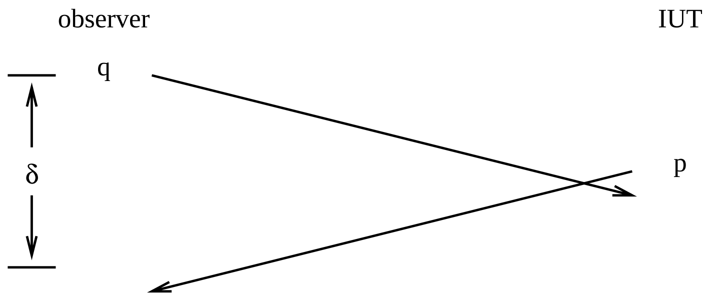

Figure 2.4: An impossible sequence of events

(ii) If $t(p) < t(q)$, then $p \prec q$.

(iii) If $t(p) > t(q) + \delta$, then $q \prec p$.

(iv) If $t(q) < t(p) < t(q) + \delta$, then $p \not\prec q$ and $q \not\prec p$.

**Sets of possible states**

Let $p_1, p_2, \ldots$ be the sequence of messages sent by the IUT, in the order observed. Let $q_1, q_2, \ldots$ be the sequence of messages sent to the IUT. We have a partial ordering $\prec$ of these messages, as described above.

Consider a sequence of messages $(m_1, \ldots m_n)$, where the $m_1, \ldots m_n$ are taken from the messages $p_1, \ldots$ and $q_1, \ldots$. We say that this sequence is *consistent* with the partial ordering $\prec$ if the following condition holds: if $m$ is in the sequence, and $m' \prec m$, then $m'$ is in the sequence, and $m'$ precedes $m$ in the sequence.

Since $p_1 \prec p_2 \prec \ldots$ and $q_1 \prec q_2 \prec \ldots$, a sequence consistent with $\prec$ which consists of the messages $p_1, \ldots p_i$ and $q_1, \ldots q_j$ will be of the form

$(\ldots p_i, \ldots q_j)$ or $(\ldots q_j, \ldots p_i)$.

We define $S_{i,j}$ to be the set of all states of the IUT which could result from a sequence consistent with $\prec$ which consists of the messages $p_1, \ldots p_i$ and $q_1, \ldots q_j$. We define $S_{0,0}$ to be $Q$, the set of all possible states.

Note that if $p_{i+1} \prec q_j$, then any sequence consistent with $\prec$ which contains $q_j$ must also contain $p_{i+1}$. Therefore, there exist no sequences consistent with $\prec$ which contain only $p_1, \ldots p_i$ and $q_1, \ldots q_j$, and $S_{i,j}$ is empty. Similarly, if $q_{j+1} \prec p_i$, then $S_{i,j}$ is empty.

For example, if $p_1 \prec q_1$, and $q_1 \prec p_2$, $S_{1,1}$ would be the set of states that could result from the sequence $(p_1, q_1)$, that is, the set $c(q_1(c(p_1(Q))))$. The sequence $(q_1, p_1)$ would not be consistent with the partial ordering.

On the other hand, if $p_2 \prec q_1$, then $S_{1,1}$ would be empty, since all sequences consistent with the partial ordering would have to begin with $(p_1, p_2, \ldots)$.

It may be helpful to consider the sets $S_{i,j}$ as forming a two-dimensional array (Figure 2.5), with the messages $p_1, p_2, \ldots$ along the left side and $q_1, q_2, \ldots$ along the top. If a square $(i, j)$, $i, j \neq 0$, is non-empty, then either the square immediately to its left or the square immediately above it must be non-empty. To see why, observe that if $(i, j)$ is not empty, there must be some sequence of the form $(\ldots p_i, \ldots q_j)$ or $(\ldots q_j, \ldots p_i)$ which is consistent with the partial ordering $\prec$. If we remove the last message in the sequence, we obtain a sequence of the form $(\ldots p_i, \ldots q_{j-1})$ or $(\ldots q_{j-1}, \ldots p_i)$—indicating that the square $(i, j-1)$ is not empty—or of the

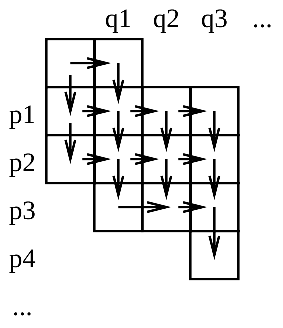

Figure 2.5: The sets $S_{i,j}$

form $(\dots q_j, \dots p_{i-1})$ or $(\dots p_{i-1}, \dots q_j)$—indicating that the square $(i-1, j)$ is not empty.

Also note that if $q_j \prec p_i$, then $S_{i,j-1}$—the square immediately to the left of $(i, j)$—is empty, because there are no sequences of the form $(\dots p_i, \dots q_{j-1})$ or $(\dots q_{j-1}, \dots p_i)$. Similarly, if $p_i \prec q_j$, then $S_{i-1,j}$ is empty.

Hopefully this discussion makes things clearer, and not more obscure. The following lemma indicates exactly how the contents of the square (i, j) can be calculated from those of its neighbors.

*Lemma 3.* (i) If $t(q_j) + \delta < t(p_i)$, then $S_{i,j} = c(p_i(S_{i-1,j}))$.

(ii) If $t(p_i) < t(q_j)$, then $S_{i,j} = c(q_j(S_{i,j-1}))$.

(iii) If $t(q_j) < t(p_i) < t(q_j) + \delta$, then $S_{i,j} = c(p_i(S_{i-1,j})) \cup c(q_j(S_{i,j-1}))$.

*Proof.* (i) By Lemma 2, we know that $q_j \prec p_i$. So $S_{i,j}$ consists of the states corresponding to sequences of the form $(\dots q_j, \dots p_i)$ consistent with $\prec$. But each of these sequences is formed by appending $p_i$ to a subsequence of the form $(\ldots q_j, \ldots p_{i-1})$ or $(\ldots p_{i-1}, \ldots q_j)$; and the set $S_{i-1,j}$ is exactly the set of all possible states resulting from these subsequences. Therefore, $c(p_i(S_{i-1,j}))$ is the set of all possible states resulting from these subsequences followed by $p_i$, which is the same as the set $S_{i,j}$.

(ii) is analogous to (i), with the $p$'s and $q$'s exchanged.

(iii) simply combines (i) and (ii).□

In other words, the square $(i,j)$ can be calculated directly from its nonempty neighbors immediately above it and to its left, as indicated by the arrows in Figure 2.5.

**The algorithm**

The following algorithm calculates the sets $S_{i,j}$, one column at a time. It uses three data structures: a list of sets $s_i$, corresponding to sets $S_{i,j}$ for a particular value of $j$; a queue of messages $p_i$ observed travelling from the IUT within the last $\delta$ time-units; and a queue of messages $q_j$ observed travelling to the IUT within the last $\delta$ time-units. Each message is kept for time $\delta$, then discarded.

*Algorithm.* (1) When $p_i$ is observed, put it on the from-IUT queue and set $s_i := c(p_i(s_{i-1}))$ (Lemma 3(i)). After time $\delta$, dequeue and discard $p_i$, and discard the set $s_{i-1}$.

(2) When $q_j$ is observed, put it on the to-IUT queue. After time $\delta$, dequeue it and calculate new sets $s'_i$ as follows. (During this step, the sets $s'_i$ correspond to the sets $S_{i,j}$; the sets $s_i$ correspond to the sets $S_{i,j-1}$.)

For the $i$ which is the index of the last message taken off the from-IUT queue, let $s'_i := c(q_j(s_i))$ (Lemma 3(ii)).

For each message $p_i$ still on the from-IUT queue, we calculate a corresponding set $s'_i$: let $s'_i := c(p_i(s'_{i-1})) \cup c(q_j(s_i))$ (Lemma 3(iii)).

If all the sets $s'_i$ are empty, fail the IUT.

Otherwise, replace the old sets $s_i$ with the new sets $s'_i$.

*Invariant.* Let $q_j$ be the last message taken off the to-IUT queue. Then if $p_i$ is the last message taken off the from-IUT queue, or if $p_i$ is still on the from-IUT queue, $s_i = S_{i,j}$.

*Proof of invariant.* (1) Suppose that the invariant holds when $p_i$ is received. Then $p_{i-1}$ is either still on the from-IUT queue (if the queue is not empty) or the last message taken off the queue (if the queue is empty), so $s_{i-1} = S_{i-1,j}$. Since $q_j$ has been dequeued, $t(q_j) + \delta < t(p_i)$, so by Lemma 3(i), $S_{i,j} = c(p_i(S_{i-1,j}))$, or $s_i = c(p_i(s_{i-1}))$. Thus the invariant is preserved.

The invariant is clearly preserved when $p_i$ is dequeued.

(2) Suppose that the invariant holds when $q_j$ is dequeued, that is, $s_i = S_{i,j-1}$ for all $i$ such that $p_i$ is the last message taken off the from-IUT queue, or is still on the from-IUT queue. We show that $s'_i = S_{i,j}$ for all such $i$, by induction.

If $p_i$ is the last message taken off the from-IUT queue, then $s_i = S_{i,j-1}$. $p_i$ was dequeued before $q_j$ was, so $t(p_i) < t(q_j)$. By Lemma 3(ii), $S_{i,j} = c(q_j(S_{i,j-1})) = c(q_j(s_i))$, and $s'_i = S_{i,j}$.

If $p_i$ is still on the queue, then (by our inductive hypothesis) $s'_{i-1} =$

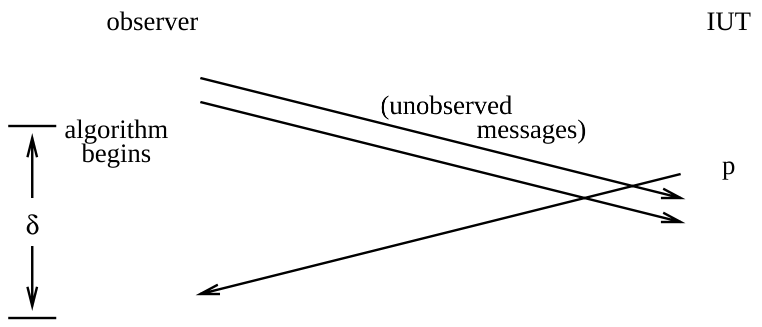

Figure 2.6: Unobserved messages at initialization

$S_{i-1,j}$. Since $p_i$ has not been dequeued yet, while $q_j$ has, $t(q_j) < t(p_i)$; and since $p_i$ was observed before $q_j$ was dequeued, $t(p_i) < t(q_j) + \delta$. By Lemma 3(iii), $S_{i,j} = c(p_i(S_{i-1,j})) \cup c(q_j(S_{i,j-1})) = c(p_i(s'_{i-1})) \cup c(q_j(s_i))$. So $s'_i = S_{i,j}$, and the invariant is preserved.$\square$

### Initialization of the algorithm

When the trace analysis algorithm is first started, there may already be messages travelling from the IUT to the observer, and vice versa (Figure 2.6). As described above, the algorithm will not be able to handle this situation properly, since it assumes that the observer sees all messages sent to the IUT.

Let $\tilde{c}(S)$ denote the completion of a set $S$ with respect to invisible events *and* messages received by the IUT: that is, the set of all states reachable from a state in $S$ via invisible transitions and messages received by the IUT. If we know that the set of possible states of the IUT includes a set $S$, and if the IUT may receive messages which are not seen by the observer, then the set of possible states of the IUT must also include the set $\tilde{c}(S)$. Any of the states in $\tilde{c}(S)$ can be reached from a state in $S$ without any messages being seen by the observer.

Suppose that, before the algorithm has been running for time $\delta$, a message $p$ sent by the IUT is observed (Figure 2.6). Then there may be messages travelling from the observer to the IUT, not seen by the observer, which are received by the IUT after it sends $p$. Therefore, until time $\delta$, the trace analysis algorithm should take the completion of all state sets with respect to both invisible transitions *and* messages received by the IUT: in other words, substitute $\tilde{c}$ for $c$ in step (1) above. After time $\delta$, the algorithm operates normally (using $c$ instead of $\tilde{c}$).

### 2.3.4 Algorithm 3: lost messages

The second algorithm assumes that the communications service used to send messages between the observer and the IUT is reliable, delivering all messages correctly. This assumption may not always be true. For example, when testing a data-link protocol, we cannot always assume that the physical layer is reliable.

We now modify the second algorithm to determine whether the observed sequence of messages could have been produced by a correct implementation, taking the possibility of messages being lost between the observer and the IUT into consideration. We will still assume, however, that messages are delivered in order, if they are delivered at all.

Messages sent by the IUT may be lost before they reach the observer. Let $\hat{c}(S)$ denote the completion of $S$ with respect to invisible transitions and messages sent by the IUT: that is, the set of states reachable from a state in $S$ via invisible transitions and transitions in which the IUT sends a message. It is analogous to $c(S)$ and $\tilde{c}(S)$. If we know that the set of possible states of the IUT includes a set $S$, and if the IUT may send messages which are not seen by the observer, then the set of possible states of the IUT must also include the set $\hat{c}(S)$.

Let $p_1, p_2, \ldots$ and $q_1, q_2, \ldots$ be the messages observed. Note that some messages sent by the IUT may be lost before reaching the observer, and hence will not appear in the sequence $p_1, p_2, \ldots$; similarly, some messages in the sequence $q_1, q_2, \ldots$ will be lost before they reach the IUT.

We now define $S_{i,j}$ to be the set of all states of the IUT which could result from a sequence of events, consistent with the partial ordering $\prec$, which contains the messages $p_1, p_2, \ldots p_i$ and possibly additional messages sent by the IUT, and does not contain any of the messages $p_{i+1}, p_{i+2}, \ldots, q_{j+1}, q_{j+2}, \ldots$. (This is similar to the previous definition of $S_{i,j}$, except that any of the messages sent to the IUT may be lost, and some of the messages sent by the IUT may not reach the observer.) We obtain a modified Lemma 3:

*Lemma 3′.* (i) If $t(q_j) + \delta < t(p_i)$, then $S_{i,j} = \hat{c}(p_i(S_{i-1,j}))$.

(ii) If $t(p_i) < t(q_j)$, $S_{i,j} = \hat{c}(q_j(S_{i,j-1})) \cup S_{i,j-1}$.

(iii) If $t(q_j) < t(p_i) < t(q_j) + \delta$, then $S_{i,j} = \hat{c}(p_i(S_{i-1,j})) \cup \hat{c}(q_j(S_{i,j-1})) \cup$

$S_{i,j-1}$.

The proof is similar to the proof of Lemma 3. In cases (ii) and (iii), we must take into consideration the possibility that $q_j$ may not be received at all. Messages sent by the IUT which do not reach the observer are taken care of by the $\hat{c}$ operator.

The algorithm is modified accordingly.

*Algorithm.* (1) When $p_i$ is observed, set $s_i := \hat{c}(p_i(s_{i-1}))$ and enqueue $p_i$. After time $\delta$, dequeue and discard $p_i$, and discard the set $s_{i-1}$.

(2) When $q_j$ is observed, enqueue it immediately. After time $\delta$, dequeue it and update the sets $s_i$ as follows.

If $p_i$ is the last message taken off the from-IUT queue, set $s'_i := \hat{c}(q_j(s_i)) \cup s_i$.

If $p_i$ is still on the from-IUT queue, set $s'_i := \hat{c}(p_i(s'_{i-1})) \cup \hat{c}(q_j(s_i)) \cup s_i$.

If all the $s'_i$ are empty, fail the IUT.

Otherwise, replace the sets $s_i$ with the newly defined sets $s'_i$.□

Initialization of the modified algorithm is somewhat simpler. From the time the algorithm is started until time $\delta$ has passed, messages may be both sent and received by the IUT without being observed by the trace analyzer; therefore, if the set of possible states is not empty, it must contain all possible states. (We assume that all states of the IUT are reachable.) Accordingly, we substitute $\bar{c}$ for $\hat{c}$ in step (1) of the algorithm until time $\delta$ has passed, where $\bar{c}(S)$ is defined to be $Q$ (the set of all possible states of the IUT) if $S$ is not empty, or the empty set if $S$ is empty.

### 2.3.5 Other variations

We have presented three trace analysis algorithms, each of which makes certain assumptions about how messages are observed. In general, there is a tradeoff between our assumptions and the errors which we can detect: if our assumptions are weaker, it becomes more difficult to be certain that the IUT has made an error.

Other variations are possible. For example, we can devise an algorithm which assumes that messages may be damaged, but are never lost. These assumptions are weaker than those of the second algorithm (messages can never be damaged), but stronger than those of the third algorithm (messages can be lost). We can also imagine trace analysis algorithms whose assumptions are even weaker than those presented here: for example, an algorithm which assumes that messages can be damaged, lost, or delivered out of order.

# Chapter 3

# A LAPB trace analyzer

Having discussed why trace analysis should be used in conformance testing, and given specific algorithms for doing so, we now demonstrate the feasibility of the proposed methodology by applying it to a specific protocol, the X.25 LAPB protocol as specified by ISO 7776 [5]. This chapter describes the design and implementation of a trace analyzer for the LAPB protocol; the following chapter discusses the effects of re-specifying the standard TTCN test suite according to the proposed methodology. Both the source code for the trace analyzer and the re-specified test suite are included in the Appendix.

## 3.1 Functionality

The LAPB trace analyzer implements all three trace analysis algorithms presented in the previous chapter; the user selects the appropriate one. The protocol specification which is used to analyze the trace is described in detail in Appendix A. Basically, it is derived from ISO 7776 by splitting receive/send transitions into two separate transitions with an intermediate state, and modelling timeouts as spontaneous events. The data transfer specification is also somewhat simplified.

The trace analyzer runs under the Unix operating system, on a Sun 3 workstation. It operates off-line, reading a recorded trace and checking it for errors. Detailed directions for its use are given in Appendix B.

## 3.2 Design and implementation

The trace analyzer is decomposed into modules according to Parnas' criterion of information hiding [14]. Each module hides a design decision from the rest of the system. In particular, all knowledge of the protocol being analyzed is confined to one module. This makes it possible to modify the protocol specification, or even to change the trace analyzer to handle a different protocol, by changing a single module. Similarly, the decision to do the trace analysis off-line instead of on-line is hidden within one module.

The trace analyzer consists of four modules. The Buffers module provides facilities for handling octet buffers of arbitrary length, used to store observed messages. The Events module hides the decision to do trace analysis off-line; the Protocol module hides the protocol specification. Finally, the main module implements the actual trace analysis algorithms.

The implementation of the trace analyzer was straightforward: it comprises about 2000 lines of C code. Appendix C provides brief descriptions of the module interfaces and the modules themselves.

## 3.3 Operation

Some examples of the trace analyzer in operation are given here. The traces being analyzed were obtained from an IDACOM PT. The PT is an X.25 protocol tester which can be used to do both monitoring and emulation. It has fairly sophisticated facilities for capturing and displaying protocol traces in various formats. In addition, user-written test scripts can be used to control the X.25 emulation (for example, to make it send an invalid message). These test scripts can be used to implement test cases.

The PT actually has two CPUs, operating independently of one another, each of which can be used to do monitoring or emulation. The traces shown here were obtained by running two X.25 emulations against one another, one on each CPU. CPU1 acts as the DCE, and CPU2 acts as the DTE; the trace analyzer checks the behavior of the DTE only. The traces are recorded at the DCE; the round-trip propagation delay between the DTE and DCE is about 35 milliseconds. In the examples shown here, 50 milliseconds is used for the parameter $\delta$.

In the first trace, the DCE and the DTE send DISC commands at the same time (DISC collision). The DCE then initiates link setup. Finally, the

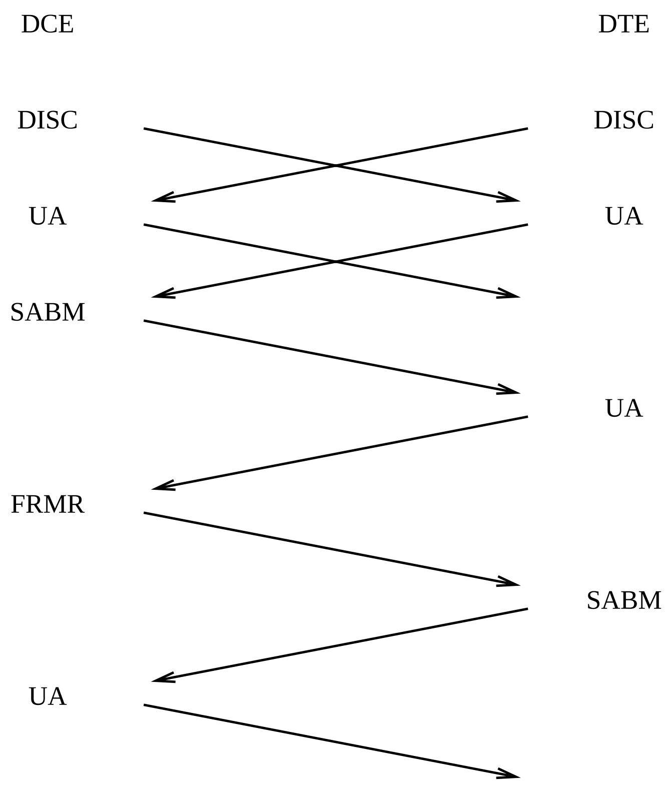

Figure 3.1: Sequence of events in trace 1

DCE requests link reset by sending an FRMR response, and the DTE resets the link. See Figure 3.1.

Analyzing it using the second algorithm of Chapter 2 (propagation delay, messages not lost), no errors are found:

```
Trace analysis begins
delta = 0.050000
DTE value of k is 7
DTE value of N1 is 1080, DCE value of N1 is 1080
```

```
12.572500 rx A / U / DISC / P/F = 1 /
12.576700 tx B / U / DISC / P/F = 1 /
12.613500 tx A / U / UA / P/F = 1 /
12.634500 rx B / U / UA / P/F = 1 /
31.957100 rx A / U / SABM / P/F = 1 /
31.964100 tx A / U / UA / P/F = 1 /
33.858600 rx B / U / FRMR / P/F = 1 / 731000
33.866700 tx B / U / SABM / P/F = 1 /
33.922900 rx B / U / UA / P/F = 1 /

Trace analysis ends, no error discovered
```

The output of the trace analyzer is fairly primitive: it simply echoes each frame as it is decoded and processed. The time at which the frame is observed is shown in the first column; it is given in seconds. The "rx" or "tx" in the second column indicates whether the frame was received or sent by the IUT. The remaining fields correspond to fields of the frame: address, type (I, S, or U), and so forth.

Note that in the trace, the DTE appears to send its DISC command *after* receiving the DISC command from the DCE, when in fact it sent its DISC *before* receiving the DCE's DISC. The second algorithm handles this situation correctly.

The second trace shows data transfer: the DCE and DTE exchange I frames. See Figure 3.2.

Again, analyzing the trace using the second algorithm, no errors are found:

```
Trace analysis begins
delta = 0.050000
```

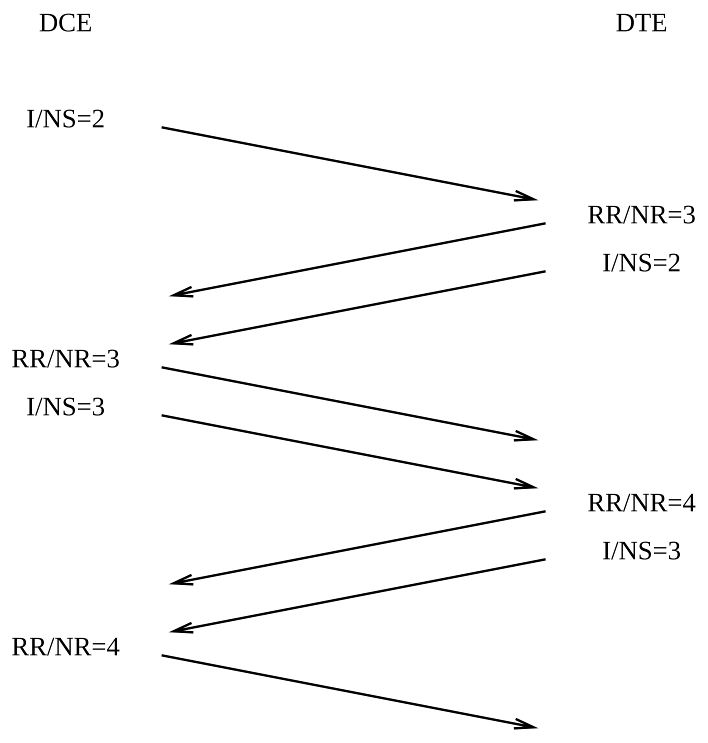

Figure 3.2: Sequence of events in trace 2

```
DTE value of k is 7
DTE value of N1 is 1080, DCE value of N1 is 1080

42.945000 rx A / I / N(S) = 2 / P = 0 / N(R) = 2 / 1001002049
4441434F4D20454C454354524F4E494353204C5444202042524 94E4753202
0544F2020594F552020544845202020502054202121212020202020202020
202020544845202050524F544F434F4C20202054455354455220205448415
420204C4541445320205448452020574159 20494E544F2054484520465554
55524520
42.970400 tx A / S / RR / P/F = 0 / N(R) = 3 /
42.980200 tx B / I / N(S) = 2 / P = 0 / N(R) = 3 / 100121
43.101900 rx B / S / RR / P/F = 0 / N(R) = 3 /
43.592000 rx A / I / N(S) = 3 / P = 0 / N(R) = 3 / 1001022049
4441434F4D20454C454354524F4E494353204C544420204252494E4753202
0544F2020594F552020544845202020502054202121212020202020202020
202020544845202050524F544F434F4C20202054455354455220205448415
420204C454144532020544845202057415920494E544F2054484520465554
55524520
43.617700 tx A / S / RR / P/F = 0 / N(R) = 4 /
43.627500 tx B / I / N(S) = 3 / P = 0 / N(R) = 4 / 100141
43.748300 rx B / S / RR / P/F = 0 / N(R) = 4 /

Trace analysis ends, no error discovered
```

In the third trace, the DCE sends a SABM to the DTE, but the DTE does not send a UA in response; instead, it immediately sends an I frame (a restart request packet). Eventually both sides time out—the DCE retransmits its SABM, the DTE sends a polling RR command—and the link is set up. See Figure 3.3.

This trace could be caused by one of two things. The first possibility is that the IUT is invalid: the first time it received the SABM, it did not send a UA before entering information transfer state. The second possibility is that the IUT is valid, and sent the UA correctly, but because the underlying

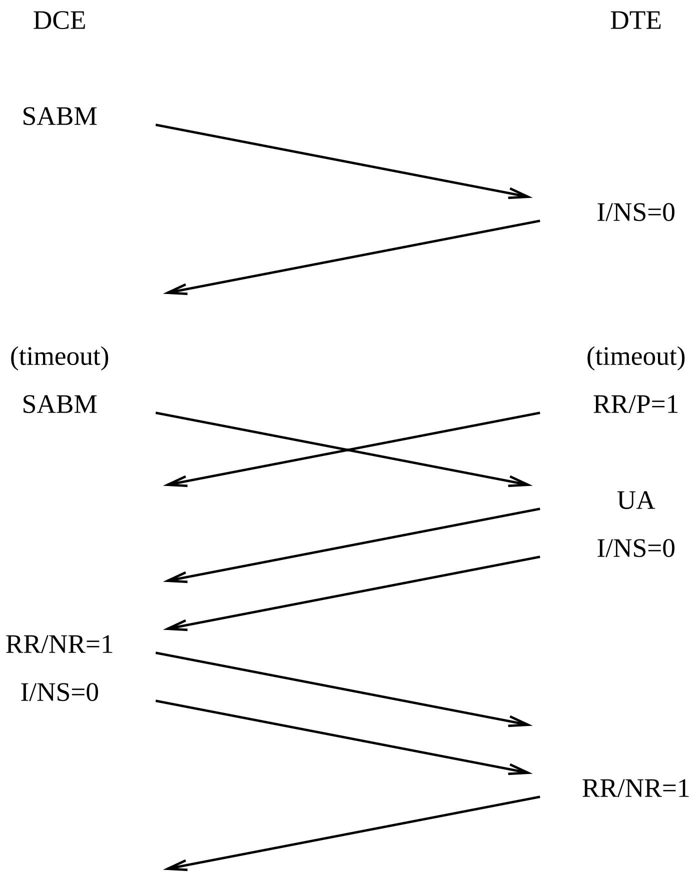

Figure 3.3: Sequence of events in trace 3

physical service is unreliable, the UA was lost between the IUT and the observer.

If we apply the second algorithm to this trace, it tells us that the IUT's behavior is invalid, because it assumes that messages are never lost.

```
Trace analysis begins
delta = 0.050000
DTE value of k is 7
DTE value of N1 is 1080, DCE value of N1 is 1080

21.456000 rx A / U / SABM / P/F = 1 /
21.465800 tx B / I / N(S) = 1 / P = 0 / N(R) = 1 / 1000FB0000
24.345600 tx B / S / RR / P/F = 1 / N(R) = 1 /

Error detected in trace at time 24.395600
```

If we apply the third algorithm, however, it tells us that the IUT is valid.

```
Trace analysis begins
delta = 0.050000
messages may be lost between the observer and the IUT
DTE value of k is 7
DTE value of N1 is 1080, DCE value of N1 is 1080

21.456000 rx A / U / SABM / P/F = 1 /
21.465800 tx B / I / N(S) = 1 / P = 0 / N(R) = 1 / 1000FB0000
24.345600 tx B / S / RR / P/F = 1 / N(R) = 1 /
24.465600 rx A / U / SABM / P/F = 1 /
24.474500 tx A / U / UA / P/F = 1 /
24.571000 tx B / I / N(S) = 0 / P = 0 / N(R) = 0 / 1000FB0000
24.587200 rx B / S / RR / P/F = 0 / N(R) = 1 /
24.665600 rx A / I / N(S) = 0 / P = 0 / N(R) = 1 / 1000FF
24.677800 tx A / S / RR / P/F = 0 / N(R) = 1 /

Trace analysis ends, no error discovered
```

Which algorithm is to be considered correct? It depends on whether the underlying physical service is, in fact, reliable or not. If the service is unreliable, and was responsible for the UA being lost, then the third algorithm is correct; the second algorithm is, in effect, blaming the IUT for a fault of the underlying physical service.

On the other hand, if the service is reliable, and it was the IUT that failed to send the UA, then the second algorithm is correct; the third algorithm fails to detect the error because it cannot be sure that the IUT did not send the UA.

# Chapter 4

# A LAPB test suite

To illustrate the advantages of the proposed methodology, we have rewritten the standard TTCN test suite for LAPB, given in ISO 8882-2, to specify only the expected behavior of the IUT in each test case. The resulting test suite, given in Appendix D, is only 8 pages long, compared to 252 in the original.

This chapter first describes the original test suite, and discusses some of the problems with it. The re-specified test suite is then discussed.

## 4.1 The original test suite

ISO 8882 [7] describes the standard conformance testing procedures for X.25 DTEs. It is divided into three parts, which are progressing towards standardization separately: ISO 8882-1 gives an overview, ISO 8882-2 specifies the data-link layer (LAPB) test suite, and ISO 8882-3 specifies the packet layer test suite. We will be referring to the November 1989 version of ISO

8882-2, JTC 1/SC 6/N 5503.

ISO 8882-2 tests the data-link layer of DTEs which are supposed to conform to CCITT X.25 1980, CCITT X.25 1984, or ISO 7776. (The requirements of these three standards are slightly different.) The test suite assumes basic sequence numbering, single link operation, DTE/DCE operation, and octet alignment.

The 287 test cases making up the test suite are divided into eight groups. There is one test group for each of seven states: disconnected phase (DL1), link disconnection (DL2), link setup (DL3), information transfer phase (DL4), frame reject condition (DL5), DTE busy condition (DL6), and sent REJ condition (DL7). Each of these seven test groups is divided into three subgroups, testing the IUT's responses to proper, improper, and inopportune frames, respectively. The eighth group, DL8, verifies that the IUT handles timeouts correctly.

All test cases are specified in TTCN; the remote test method is used. A typical test case consists of a preamble to put the IUT in the appropriate state, a frame sent by the tester to the IUT, an expected response from the IUT, and a postamble to verify that the IUT is in the correct state. There may be more than one possible response, and therefore more than one postamble used. Some test cases, particularly in the data transfer states DL4, DL6, and DL7, use attached subtrees to handle unexpected but valid events (such as unexpected I or RR frames from the IUT).

ISO 8882-2 gives possible preambles and postambles for each of the seven states, but indicates that they are only intended as examples. Other preambles and postambles may be used by agreement between the test laboratory and the client.

## 4.2 Problems with the original test suite

The ISO 8882-2 test suite has a number of major problems. We can group them into five areas.

First, the test suite standard itself is extraordinarily large. ISO 7776, the protocol standard from which the test suite is derived, is 23 pages long; ISO 8882-2 is 252 pages long, an entire order of magnitude larger. LAPB is a relatively simple protocol; one wonders how long a test suite for the transport or session layer protocols would be. It is very difficult to read and understand a document of this size, to detect errors in it, and to make changes to it.

Second, the test suite makes some questionable assumptions. It is stated that "DISC and SABM commands and DM responses sent by the IUT are not considered to be acceptable unexpected frames during information transfer tests"; but no real justification is given for this statement. Moreover, the assumption that the IUT will not send DISC, SABM or DM frames is not limited to the information transfer tests; for example, the test cases in group DL1 make this assumption as well. This assumption does not seem justifiable: there certainly exist DTEs which send DISC, SABM, and DM frames spontaneously. This behavior is permitted by the original protocol standards, but will cause the DTE to fail the standard conformance tests.

It is also assumed that frames are never lost by the underlying physical layer; this assumption is not stated explicitly by ISO 8882-2. This assumption may be justifiable, provided the physical connection between the tester and the IUT is reliable enough. However, if the physical connection does happen to lose a frame, the IUT will probably fail the current test case, through no fault of its own.

These assumptions simplify the test cases, by reducing the number of unexpected but valid events. Changing the test suite to do without these assumptions would make it even larger than it is already.

A third problem is that those unexpected but valid events which *are* handled—namely I, RR, RNR, and REJ frames during information transfer tests—are handled in a rather clumsy and limited way. For example, the NORMAL_INFORMATION_TRANSFER test step, which is used by many test cases, checks that the N(S) field in I frames sent by the IUT is correct, but only checks that the N(R) field acknowledges either all I frames sent by the tester, or all I frames but one. If the tester hasn't sent any I frames at all, and the IUT happens to send an I frame with N(R)=7—which is invalid—the tester will accept it. The N(R) field in RR, RNR, and REJ frames is not checked at all.

Fourth, the current version of ISO 8882-2 contains many, many errors, despite repeated revisions over several years by protocol experts. There are numerous errors in TTCN usage. In some test cases, for example DL2_110, the test suite does not indicate what the tester should do if a boolean condition does not hold. In test cases which attach the test steps ACCEPTABLE_UNEXPECTED_DL4, _DL6, and _DL7, there is no GOTO back to the top of the list of alternatives.

Several test cases in DL4 do not handle unexpected but valid I-frames or supervisory frames.

As already mentioned, unexpected but valid SABM, DISC, and DM frames are not handled. If the IUT sends one of these frames, it will fail.

Several cases, for example DL4_105, will only accept RR or RNR responses, not commands.

The test step LINK_SET_UP is attached inappropriately in a number of cases, for example DL4_201: if the timer elapses without a frame being received from the IUT, the verdict PASS will be assigned, instead of FAIL.

We have found many other errors in various test cases; and undoubtedly there are many which we haven't found.

And finally, as a whole the test suite seems remarkably disorganized. The numbering and ordering of the test cases is strange. For example, in test group DL1, there are test cases 201A, 201B, 202 to 205 (but not 206), 207 to 211, 215 and 216. What happened to the missing test cases? There seems to be no discernible reason for the ordering of test cases. There are several test cases which have two test case references (for example, DL2_111); this is never explained. There seems to be unnecessary repetition in the test suite: in DL3, for example, the IUT reaction to RR, RNR, REJ frames with P=0, P=1, F=0, F=1—namely, discard—is tested; 12 test cases in all. Finally, in the test step library, there are some instances where two quite different test steps have the same stated test purpose; for example, NORMAL_INFORMATION_TRANSFER and ACCEPTABLE_UNEXPECTED_DL4. This disorganization contributes to the difficulty of reading and understanding the test suite.

In short, the ISO 8882-2 test suite has a number of major problems, which do not appear likely to be solved in the near future.

## 4.3 The re-specified test suite

In order to demonstrate the advantages of the proposed methodology, we have re-specified the ISO 8882-2 test suite, having each test case only specify the *expected* IUT behavior; any unexpected behavior will be checked for validity by the trace analyzer.

Most test cases in the test suite consist of one stimulus sent to the IUT and a set of possible responses from the IUT, and can be specified on one line. The test suite has been changed slightly to test ISO 7776 DTEs exclusively, not CCITT X.25 1980 or 1984 DTEs, but the differences between the three standard are minor and affect only a few test cases. The same test suite structure and same test case numbering is used; inconsistencies with the protocol standards are not corrected. No variables are used: in the data transfer tests, sequence numbers are specified explicitly, assuming that the DL4, DL6, and DL7 preambles ensure the IUT's V(S) and V(R) variables are set to 0. Preambles and postambles are not given; procedures which are mutually agreeable between the test laboratory and the client should be used.

The complete test suite is included in Appendix D. It consists of only 8 pages, compared to 252 in the original. In our opinion, it has a number of major advantages over the original. Because the test cases are so much simpler, and because the test suite is so much smaller, the test suite is much easier to read and understand; inconsistencies with the protocol standard become more visible; and changes may be made quickly. The overall reliability and usefulness of the LAPB conformance testing procedure are thereby enhanced.

## 4.4 Implementing the test cases

Implementing the test cases is straightforward. The typical test case implementation first executes a preamble to put the IUT in the appropriate state, then sends the specified stimulus to the IUT, and starts a timer (the timer's period is another matter for agreement between the test laboratory and the client). If the expected response is received before the timer expires, a postamble is executed to check that the IUT is in the correct state. If some unexpected message is received, or the timer expires, the test is aborted. In the data transfer test cases, unexpected information and supervisory frames may be received from the IUT without affecting the test results; these frames are simply discarded. The trace analyzer will check their validity.

Assigning a verdict is simple enough. If the trace is invalid (determined by the trace analyzer), the IUT fails. If the trace is valid, and the expected behavior is observed, the IUT passes. If the trace is valid, but the expected behavior was not observed, the verdict is inconclusive.

We have implemented a couple of representative test cases, DL1_101 and DL4_108, on the IDACOM PT. Some traces from execution of the test cases are shown below. For each test case, three traces are shown, corresponding to (a) the expected IUT behavior, (b) invalid IUT behavior, and (c) unexpected but valid IUT behavior. Note that the test cases are not expected to distinguish between (b) or (c); the trace analyzer does this.

### 4.4.1 Test case DL1_101

DL1_101 verifies that the IUT sends a DM with F=1 in response to a DISC command with P=1 received in the disconnected state. In the re-specified test suite, DL1_101 can be specified on a single line (the message sent by the tester is shown on the left, the IUT's response on the right):

```
DISC/P=1                            DM/F=1
```

The specific preambles and postambles used to ensure that the IUT is in a particular state or to verify that the IUT is in a particular state are left up to the test case implementor. In this case, we send a DISC and wait for a DM or UA as a preamble; we send an RR command with P=1 and wait for a DM with F=1 as a postamble.

The following trace, in a format produced by the IDACOM PT, shows the expected IUT behavior. Frames marked "DCE" indicate frames sent by the tester (on CPU1); frames marked "DTE" indicate frames sent by the IUT (the emulation on CPU2).

```
DL1_101:  preamble
09:54.5087  DCE  ADDRESS=03  FRAME=DISC   P=1
09:54.5090
09:54.5161  DTE  ADDRESS=03  FRAME=DM     F=1
09:54.5164
DL1_101:  test body
09:54.6899  DCE  ADDRESS=03  FRAME=DISC   P=1
09:54.6902
09:54.6969  DTE  ADDRESS=03  FRAME=DM     F=1
09:54.6972
DL1_101:  postamble
09:54.7170  DCE  ADDRESS=03  FRAME=RR     P=1   NR=0
09:54.7174
09:54.7242  DTE  ADDRESS=03  FRAME=DM     F=1
09:54.7245
DL1_101:  expected behavior observed, test ends
```

The next trace shows invalid IUT behavior. In the test body, the IUT sends a UA in response to the tester's DISC command, instead of a DM.

```
DL1_101:  preamble
11:52.8495  DCE  ADDRESS=03  FRAME=DISC   P=1
11:52.8498
11:53.3393  DTE  ADDRESS=03  FRAME=DM     F=1
11:53.3396
DL1_101:  test body
```

```
11:53.3610  DCE  ADDRESS=03  FRAME=DISC   P=1
11:53.3614
11:54.0791  DTE  ADDRESS=03  FRAME=UA     F=1
11:54.0794
DL1_101:  unexpected event, test aborted
```

The last trace shows valid but unexpected behavior: the IUT sends a DM response with F=0, when the tester is expecting a DM response with F=1. This is valid, because the IUT could have sent the DM with F=0 (to request link setup) before receiving the tester's DISC command; but it is not what is expected by the test case.

```
DL1_101:  preamble
14:58.6835  DCE  ADDRESS=03  FRAME=DISC   P=1
14:58.6839
14:59.7426  DTE  ADDRESS=03  FRAME=DM     F=1
14:59.7429
DL1_101:  test body
14:59.7644  DCE  ADDRESS=03  FRAME=DISC   P=1
14:59.7648
15:00.5867  DTE  ADDRESS=03  FRAME=DM     F=0
15:00.5870
DL1_101:  unexpected event, test aborted
```

### 4.4.2 Test case DL4_108

DL4_108 verifies that the IUT handles a REJ command with P=1 correctly, by sending an RR response with F=1 and then retransmitting the requested I frames.

```
     I/P=0, NS=0, NR=0              ->
                                    <- I/NS=0
     I/P=0, NS=1, NR=0              ->
```

```
                                       <- I/NS=1
     I/P=0, NS=2, NR=0                 ->
                                       <- I/NS=2
     REJ/P=1, NR=1                     ->
                                       <- RR/F=1, NR=3
                                       <- I/NS=1, NR=3
     RR/NR=2, F=received P bit         ->
                                       <- I/NS=2, NR=3
     RR/NR=3, F=received P bit         ->
```

As this is a data transfer test case, any unexpected information or supervisory frames are simply discarded; they will be checked for validity by the trace analyzer.

As a preamble, the tester sends a DISC and waits for a UA or DM, then sends a SABM and waits for a UA. As a postamble, the tester sends an FRMR and waits for a SABM or DM.

The following trace shows the expected IUT behavior:

```
DL4_108:  preamble
26:48.5894  DCE  ADDRESS=03  FRAME=DISC   P=1
26:48.5898
26:48.5984  DTE  ADDRESS=03  FRAME=UA     F=1
26:48.5897
26:48.7674  DCE  ADDRESS=03  FRAME=SABM   P=1
26:48.7677
26:48.7760  DTE  ADDRESS=03  FRAME=UA     F=1
26:48.7763
DL4_108:  test body
26:48.7986  DCE  ADDRESS=03  FRAME=INFO   P=0   NR=0   NS=0
                 GF=1 D=0 Q=0 LCN=0   RESTART INDICATION PACKET
                 CAUSE      = 00 UNDEFINED
                 DIAGNOSTIC = 00 NO ADDITIONAL INFORMATION
26:48.7996
26:48.8146  DTE  ADDRESS=03  FRAME=RR     F=0   NR=1
```

```
26:48.8149
26:48.8913  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=1   NS=0
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
26:48.8920
26:48.9116  DCE  ADDRESS=03  FRAME=INFO   P=0   NR=0   NS=1
                 GF=1 D=0 Q=0 LCN=0   RESTART INDICATION PACKET
                 CAUSE      = 00 UNDEFINED
                 DIAGNOSTIC = 00 NO ADDITIONAL INFORMATION
26:48.9126
26:48.9262  DTE  ADDRESS=03  FRAME=RR     F=0   NR=3
26:48.9265
26:49.0028  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=2   NS=1
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
26:49.0035
26:49.0231  DCE  ADDRESS=03  FRAME=INFO   P=0   NR=0   NS=2
                 GF=1 D=0 Q=0 LCN=0   RESTART INDICATION PACKET
                 CAUSE      = 00 UNDEFINED
                 DIAGNOSTIC = 00 NO ADDITIONAL INFORMATION
26:49.0241
26:49.0377  DTE  ADDRESS=03  FRAME=RR     F=0   NR=3
26:49.0380
26:49.1145  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=3   NS=2
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
26:49.1152
26:49.1341  DCE  ADDRESS=03  FRAME=REJ    P=1   NR=1
26:49.1345
26:49.1433  DTE  ADDRESS=03  FRAME=RR     F=1   NR=3
26:49.1436
26:49.1530  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=3   NS=1
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
26:49.1537
26:49.1775  DCE  ADDRESS=01  FRAME=RR     F=0   NR=2
26:49.1779
26:49.1652  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=3   NS=2
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
26:49.1658
26:49.2024  DCE  ADDRESS=01  FRAME=RR     F=0   NR=3
26:49.2028
DL4_108:  postamble
```

```
26:49.2161  DCE  ADDRESS=01  FRAME=FRMR   F=0
                 CONTROL = 64      COMMAND  VR=3   VS=3   W=0 X=0 Y=0 Z=0
                 BITS 17-24 INVALID COMBINATION
26:49.2169
26:49.2284  DTE  ADDRESS=01  FRAME=SABM   P=1
26:49.2287
26:49.2371  DCE  ADDRESS=01  FRAME=UA     F=1
26:49.2375
DL4_108:  expected behavior observed, test ends
```

The following trace shows invalid IUT behavior. When the tester sends the polling REJ command to the IUT, it retransmits the requested I frames immediately, instead of sending an RR response.

```
DL4_108:  preamble
40:07.6454  DCE  ADDRESS=03  FRAME=DISC   P=1
40:07.6458
40:08.0611  DTE  ADDRESS=03  FRAME=UA     F=1
40:08.0614
40:08.0772  DCE  ADDRESS=03  FRAME=SABM   P=1
40:08.0776
40:09.0393  DTE  ADDRESS=03  FRAME=UA     F=1
40:09.0395
DL4_108:  test body
40:09.0618  DCE  ADDRESS=03  FRAME=INFO   P=0   NR=0   NS=0
                 GF=1 D=0 Q=0 LCN=0   RESTART INDICATION PACKET
                 CAUSE      = 00 UNDEFINED
                 DIAGNOSTIC = 00 NO ADDITIONAL INFORMATION
40:09.0628
40:11.7576  DTE  ADDRESS=03  FRAME=RR     F=0   NR=1
40:11.7579
40:11.7862  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=1   NS=0
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
40:11.7869
40:11.8065  DCE  ADDRESS=03  FRAME=INFO   P=0   NR=0   NS=1
                 GF=1 D=0 Q=0 LCN=0   RESTART INDICATION PACKET
                 CAUSE      = 00 UNDEFINED
                 DIAGNOSTIC = 00 NO ADDITIONAL INFORMATION
40:11.8075
40:13.4801  DTE  ADDRESS=03  FRAME=RR     F=0   NR=3
40:13.4803
40:13.5087  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=2   NS=1
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
40:13.5094
40:13.5291  DCE  ADDRESS=03  FRAME=INFO   P=0   NR=0   NS=2
                 GF=1 D=0 Q=0 LCN=0   RESTART INDICATION PACKET
                 CAUSE      = 00 UNDEFINED
                 DIAGNOSTIC = 00 NO ADDITIONAL INFORMATION
40:13.5301
40:14.8955  DTE  ADDRESS=03  FRAME=RR     F=0   NR=3
40:14.8958
40:14.9241  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=3   NS=2
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
40:14.9248
40:14.9438  DCE  ADDRESS=03  FRAME=REJ    P=1   NR=1
40:14.9442
40:16.4951  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=3   NS=1
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
40:16.4957
40:16.5263  DTE  ADDRESS=01  FRAME=INFO   P=0   NR=3   NS=2
                 GF=1 D=0 Q=0 LCN=0   RESTART CONFIRM PACKET
40:16.5270
DL4_108:  unexpected event, test aborted
```

The final trace shows an unexpected but valid sequence of events: when the tester sends an I frame to the IUT, the IUT acknowledges it, but does not send an I frame of its own. This is perfectly valid, but does not allow the test case to be performed. Eventually the tester times out and aborts the test.

```
DL4_108:  preamble
41:21.8252  DCE  ADDRESS=03  FRAME=DISC   P=1
41:21.8255
```

```
41:23.8857  DTE  ADDRESS=03  FRAME=UA     F=1
41:23.8859
41:23.9017  DCE  ADDRESS=03  FRAME=SABM   P=1
41:23.9021
41:24.6081  DTE  ADDRESS=03  FRAME=UA     F=1
41:24.6084
DL4_108:  test body
41:24.6306  DCE  ADDRESS=03  FRAME=INFO   P=0   NR=0   NS=0
                 GF=1 D=0 Q=0 LCN=0   RESTART INDICATION PACKET
                 CAUSE      = 00 UNDEFINED
                 DIAGNOSTIC = 00 NO ADDITIONAL INFORMATION
41:24.6316
41:27.6161  DTE  ADDRESS=03  FRAME=RR     F=0   NR=1
41:27.6164
DL4_108:  unexpected event, test aborted
```

# Chapter 5

# Conclusions

We have shown that the problems associated with existing TTCN test suites—excessive size and complexity, leading to large numbers of errors—may be attributed to the conventional conformance testing methodology, in which each test case must specify all possible valid sequences of events.

We propose a new methodology, in which an independent trace analyzer is used to decide whether or not the observed behavior of the IUT is valid. In this methodology, test cases only need to specify the *expected* sequence of events, since unexpected events will be checked by the trace analyzer. This simplifies the test cases dramatically. The resulting test suites are vastly smaller and simpler than the existing ones, making it much easier to detect and correct errors in them, and allowing us to place more confidence in the results which they produce. We believe that adopting the proposed methodology will greatly enhance the reliability and usefulness of OSI conformance testing.

In order to use the new methodology, trace analyzers are needed. Existing trace analysis methods turn out to be unsuitable for conformance testing. Three new trace analysis algorithms have been presented and proved.

Finally, the methodology has been applied successfully to the LAPB protocol. A LAPB trace analyzer has been designed and implemented, and the standard TTCN test suite for LAPB has been re-specified, reducing its size from 252 pages to 10 pages.

A number of areas have not been addressed in this thesis, but appear to be worthwhile to investigate in the future.

1. Other trace analysis algorithms. The algorithms presented here assume that the protocol being tested can be modelled by an extended finite state machine, and do not handle timeouts very cleanly. If the proposed methodology does become widely used, more sophisticated trace analysis algorithms—perhaps employing multiple passes—would be useful.

2. Testing of higher layers, and of more complex protocols. LAPB is a data-link protocol, and is rather simple compared to the ISO transport or session protocols.

3. Performance. In this thesis, performance was not considered an important factor. The LAPB trace analyzer discussed in Chapter 3 was not intended for on-line use, and in fact is probably too slow.

# Bibliography

[1] G. v. Bochmann and O. Bellal, "Test result analysis in respect to formal specifications," in *Proceedings of the Second International Workshop on Protocol Test Systems*, pp. 272-294, October 1989.

[2] G. v. Bochmann, R. Dssouli, and J. R. Zhao, "Trace analysis for conformance and arbitration testing," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1347-1356, November 1989.

[3] R. M. S. Cork, "The testing of protocols in SNA products—an overview," in *Proceedings of the IFIP WG 6.1 Third International Workshop on Protocol Specification, Testing, and Verification*, pp. 455-463, June 1983.

[4] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, Massachusetts: Addison-Wesley, 1979.

[5] ISO 7776, *Information processing systems — Open Systems Interconnection — Description of the X.25 LAPB-compatible DTE data link procedures.*

[6] ISO 8208, *Information processing systems — Open Systems Interconnection — X.25 packet level protocol for data terminal equipment.*

[7] ISO 8882, *Information processing systems — Open Systems Interconnection — X.25 DTE conformance testing.*

[8] ISO 9646, *Information processing systems — Open Systems Interconnection — OSI conformance testing methodology and framework.*

[9] B. Kanungo, L. Lamont, R. L. Probert, and H. Ural, "A useful FSM representation for test suite design and development," in *Proceedings*

*of the IFIP WG 6.1 Sixth International Workshop on Protocol Specification, Testing, and Verification*, pp. 163-176, June 1986.

[10] J. K.-H. Lo, "Open Systems Interconnection Passive Monitor," M. Sc. thesis, Department of Computer Science, University of British Columbia, 1990.

[11] R. S. Matthews, K. H. Muralidhar, and S. Sparks, "MAP 2.1 conformance testing tools," *IEEE Transactions on Software Engineering*, vol. 14, no. 3, pp. 363-374, March 1988.

[12] R. E. Miller, "Protocol specification: The first ten years, the next ten years; some personal observations," in *Proceedings of the IFIP WG 6.1 Tenth International Symposium on Protocol Specification, Testing, and Verification*, June 1990.

[13] R. Molva, M. Diaz, and J. M. Ayache, "Observer: A run-time checking tool for local area networks," in *Proceedings of the IFIP WG 6.1 Fifth International Workshop on Protocol Specification, Testing, and Verification*, pp. 495-506, June 1985.

[14] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.

[15] R. L. Probert, "Towards a knowledge-based model for conformance test results analysis," in *Proceedings of the First International Workshop on Protocol Test Systems*, October 1988.

[16] R. L. Probert, H. Ural, and M. W. A. Hornbeek, "An integrated software environment for developing and validating standardized conformance tests," in *Proceedings of the IFIP WG 6.1 Eighth International Symposium on Protocol Specification, Testing, and Verification*, pp. 87-98, June 1988.

[17] D. P. Sidhu and T.-K. Leung, "Formal methods for protocol testing: A detailed study," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 413-426, April 1989.

[18] A. S. Tanenbaum, *Computer Networks*, 2nd ed., Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

[19] H. Ural and R. L. Probert, "Step-wise validation of communication protocols and services," *Computer Networks and ISDN Systems*, vol. 11, no. 3, pp. 183-202, March 1986.

[20] A. Wiles, "ITEX, system for editing tests in TTCN," Technical Report, Swedish Institute of Computer Sciences, 1987.