Automatic Test Result Analysis for High-Level Specifications[†]

O.B. Bellal, G.v. Bochmann, M. Dubuc[*], F. Saba[**]

Université de Montréal
Dépt. d'I.R.O.
C.P. 6128, Succ. "A"
Montréal, Québec, Canada H3C 3J7
E-mail:bellal@iro.umontreal.ca
Tel:(514) 343-3504

Abstract

Formal specifications are intended to be used in many activities of the software development life cycle in order to increase confidence that a specified system will behave properly. Among these activities, this paper focuses on the analysis of test results in the context of OSI communication protocols. The paper discusses the principles involved in the comparison of test results with respect to a reference specification which may be non-deterministic. The analysis tool TETRA is presented which performs such analysis for specifications written in the formal description technique LOTOS. The paper also gives an overview of two experiments where TETRA was used for the testing of an OSI Application layer protocol (namely ACSE) and for the validation of the verdicts of standardized test cases for the X.25 protocol.

Keywords: Communication protocols, conformance testing, formal specifications, test result analysis, error diagnosis.

1. Introduction

A well-known problem in system testing is the realization of a reference, sometimes called "oracle", which determines whether a given interaction sequence observed during the test of an implementation under test (IUT) is valid or not. Such an oracle must clearly be related to

† Work supported by the IDACOM-NSERC-CWARC Chair of Industrial Research in Communication Protocols.

* M. Dubuc is now with IDACOM, Montreal.

** F. Saba is now with the Computer Research Institute of Montreal (CRIM).

the specification of the IUT. This paper deals with the construction of such an oracle based on a given formal specification of the IUT.

In the area of communication protocol development and implementation, the protocol specification has an important role to play [Boch 90g]. It is the basis for the protocol implementations in different systems which are designed to communicate with one another, and helps for the selection of test cases and the analysis of test results. Several so-called formal description techniques (FDT's) have been proposed for the development of formal specifications of the OSI communication protocols and services [Boch 90g]. One of these techniques, called LOTOS [Loto 89, Bolo 87], is considered in Section 3 of this paper.

In order to simplify the verification that a given communications product satisfies the requirements defined by the OSI standards, the standardization committees responsible for OSI are also developing standards about the general methodology of conformance testing and specific test suites which should be applied to check that a given IUT satisfies all conformance requirements defined by a protocol standard [Rayn 87]. A test suite essentially consists of a number of individual test cases, which are to be executed with the given IUT. Each test case defines a sequence of inputs to be applied and usually foresees various responses from the IUT; for each response either further inputs are specified or a verdict is given which indicates whether the IUT has passed the test. These verdicts embody the oracle function.

There are, however, several situations where the standardized conformance test cases are not suitable, such as the following:

(a) For the complete testing of an implementation, there are usually system-specific implementation requirements to be verified, in addition to the requirements of the protocol standard. This implies additional test cases.

(b) Sometimes it is important to execute unexpected test cases, possibly generated randomly.

(c) During interoperability testing, as shown in Figure 1.1, there are no given test cases; instead the two systems exchange messages according to a given application. If the two systems exhibit difficulties, it is the task of the arbiter to determine which of the two systems does not follow the given protocol standards.

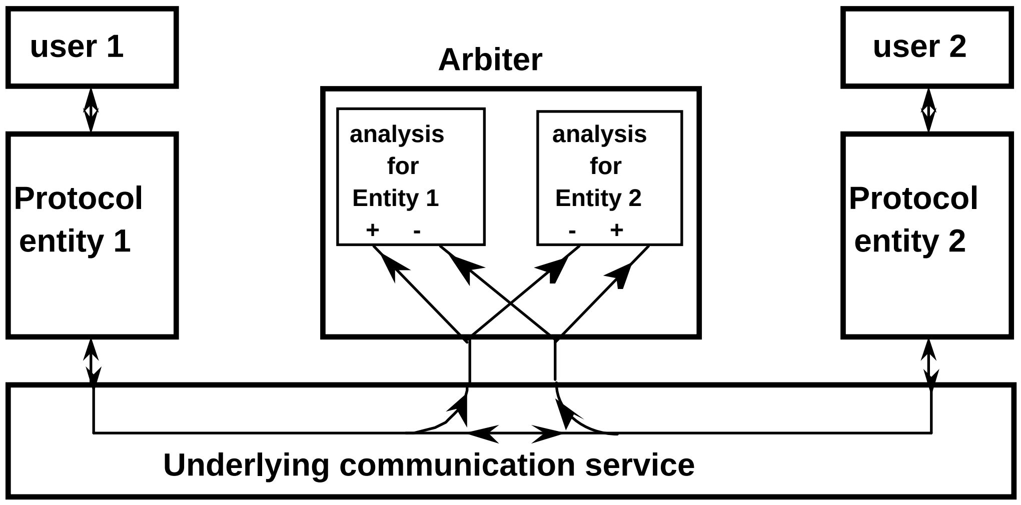


Figure 1.1: Test architecture with arbiter

In all the above cases, the oracle function is not provided. While in the first case, the possible expected outputs and related verdicts can be statically determined from the protocol and system-specific requirements, in the other cases, the oracle function must be provided dynamically, that is, the trace of interactions observed during the test must be compared with the reference specification which must be satisfied. This comparison is the topic of this paper.

In general, the following three aspects characterize the testing process :

(1) The test architecture determines through which points (interfaces) the IUT can be controlled and observed.
(2) The test suite (set of test cases) determines which kinds of faults will be detected. Sometimes the tests are selected in order to satisfy certain fault coverage criteria [Boch 91d] (for a review of test selection methods for communication protocols, see [Sari 89c]).
(3) The test result analysis checks whether the observed test trace satisfies the requirements of the reference specification. If an error is detected, it may also provide diagnostic information in order to locate the problem.

We think that it is in general useful to deal with these three aspects separately. In this paper we concentrate on the third aspect. There are, however, certain relations between the different aspects. In particular, the test architecture has an impact on the test result analysis. For instance, an architecture with partial observation implies reduced fault detection power, as discussed in [Boch 89m]. The architectural structure must also be taken into account in the determination of the reference specification if the points of control and observation do not correspond directly with the interfaces used in the specification of the IUT [Boch 89m]. In this case the reference specification used for test result analysis is different from the specification of the IUT (see also Section 4).

In the area of communication protocols, several trace analyzers have been described in the literature. They are also called trace checkers or observers. In most cases, each analyzer is built with a special protocol in mind, such as SNA [Cork 83], MAC protocol [Molv 85], class 4 Transport protocol [Matt 88] and X.25 [Prob 88]. More general trace analysis algorithms are described in [Jard 83b] and [Ural 86]. Different formalisms are used to model the reference specification of the protocol to be tested, such as EFSM, Petri Net, rule-based and knowledge-based techniques. Data cannot be represented by Petri Nets, only the control structures can be taken into account by a Petri Net based trace analyzer. Also, since a knowledge-based analyzer uses heuristic rules, it requires double checking by a human protocol expert. Protocol specifications are not taken as parameters by any of the above described analyzers, and they are not used directly as references. In each analyzer, the specification is written in a programming language such as C, Pascal or Prolog. In this situation, the adaptation of an analyzer to another protocol may not be easy to achieve without major changes in the source code. Also, while programming languages are usually deterministic, nondeterminism in a protocol specification must be handled explicitly. None of the above trace analyzers provides any diagnostic facility in case of an invalid trace. This facility would be useful to identify the reason of failure.

In Section 2 we discuss in detail how a given test trace can be validated against a reference specification. While this is relatively straightforward if the reference specification is deterministic, it is more difficult if the specification, for a given trace of observed interactions, allows for several execution histories to explain this trace. The application of these principles for the construction of a trace analysis tool for LOTOS specifications, called TETRA, is described in Section 3.

It is important to note that the principle of trace analysis can also be used to validate the verdicts of test cases with respect to the specification. In fact, a version of TETRA has been developed which allows the analysis of test cases, written in LOTOS, which may contain several branches corresponding to different responses obtained from the IUT. Each branch containing a verdict is compared with the reference specification in order to check that the verdict corresponds to the requirements of the specification.

Two experiments have been performed with trace analysis and test case validation performed by TETRA. An overview of the experiments is given in Section 4. In one case, the trace of interactions exchanged between two interworking ACSE protocol entities were analyzed with respect to the protocol specification. ACSE (Association Control Service Element) is an OSI Application layer protocol and involves ASN.1-encoded messages. The transformation between this coding scheme and the LOTOS format of interaction parameters accepted by TETRA was solved through the automatic generation of corresponding (en-)coding routines. Another experiment concerns the validation of the X.25 (LAP-B) test cases standardized by ISO and CCITT. These test cases are validated by TETRA with respect to a LAP-B specification written in LOTOS.

2. Test result analysis with respect to high-level specifications

This section explores trace analysis with respect to a reference specification written in a high-level specification language. An observed trace T is valid with respect to the reference specification if the latter allows for an execution sequence equal to the given trace. In the case that the reference specification is executable, the validity of a given trace can be easily checked by executing the reference specification using the input interactions of the trace as input to the execution. The outputs obtained during the execution should then be equal, in type and order, to the output interactions of the given trace. However, in the case that the reference specification allows for non-determinism, the checking of a given trace is more complicated since all possible execution histories of the reference specification must be considered. The trace is valid if at least one of these histories is equal to the trace.

This section gives an introduction to the specification formalism used, and discusses the issues of trace analysis in a general context. The subsequent section describes a trace analysis tool, called TETRA, which allows the validation of traces with respect to reference specifications written in LOTOS.

2.1. A high level specification formalism

We consider in this section a specification formalism which uses rendezvous interactions for communication between concurrent processes, similar as in CSP, CCS or LOTOS. We assume that the behavior of a system is expressed in terms of communicating processes. A process can be viewed as a black box able to communicate with its environment. The communication between (two or more) processes is done by rendezvous through gates, and may involve the exchange of parameters.

Interactions that occur at gates which are shared by a process with the environment of the system are called external actions. They contribute to the external behavior of the specified system. In addition, interactions that occur at the gates which are shared among the processes of the system description, and which are hidden from the environment, are called internal actions. They are not visible and do not contribute to the trace of observable interactions.

The behavior of a process is defined in terms of *behavior expressions*. They describe the order in which internal and external actions should happen. They are built using a number of operators that allow to express such concepts as alternate choice "[]", interleaved parallel composition "|||", coupled parallel composition "|[ ]|", sequential composition ";", and disabling "[>". We use here and in the following the syntax of LOTOS. A simple example of a specification involving five external gates a,b,c,d, and e, is shown as Example 2.1 below. The possible execution histories are shown in Figure 2.1 in the form of a tree. The system starts in the top node and follows one of the paths through the tree. The three branches of the tree correspond to the three choices of the behavior of process P.

Example 2.1:

```
specification   S [a,b,c,d,e] : noexit :=

behavior
        P[a,b,c,d,e]
where
        process  P[a,b,c,d,e] : noexit :=
                a; (c; e; stop
                    []
                    b; c; stop)
                []
                a; b; d; stop
        endproc

endspec
```

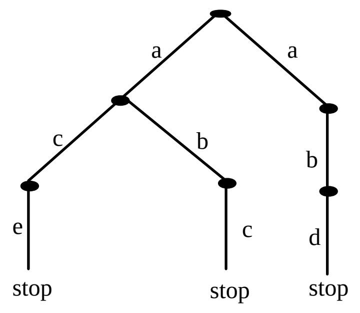


Figure 2.1: Tree representation of the specification

2.2. Search strategies for non-deterministic specifications

A specification is non-deterministic if, for a particular state, it allows for the execution of more than one action (internal or external). This relates to one of the following cases:

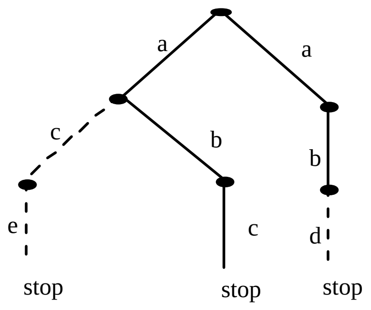
(1) For a given external action, there are more than one branch in the execution tree (see for example, the initial state in Figure 2.1).

(2) Execution of internal events (the spontaneous event i, or hidden actions) appear as alternatives within a choice of subexpressions.

Usually, non-determinism is introduced in the specification of a protocol or a service for the following reasons: a) the description of several alternatives from which one can be chosen by the implementation; b) the interleaving of actions of two or more protocol entities within the same specification; c) the use of spontaneous events for the description of events that may occur any time, such as time-out mechanisms or failures of system components.

For trace analysis, non-determinism means that, at some stage during the analysis, several paths in the reference specification are possible, and the trace analyzer has to explore all these possibilities before concluding that the given trace is invalid [Jard 83b]. The way this exploration is done depends on the search strategy implemented by the trace analyzer, and this has an important impact on the analysis performance.

As example, Figures 2.2 (a) and (b) show the projections of traces T1= "a;b;c" and T2="a;e;d", respectively, upon the behavior tree of the specification shown in Figure 2.1. One sees that trace T1 has a corresponding branch in the specification with a point of non-determinism. But trace T2 is compatible with the specification only for its first interaction. T1 is valid, while T2 is invalid since the specification does not allow the interaction e after a.



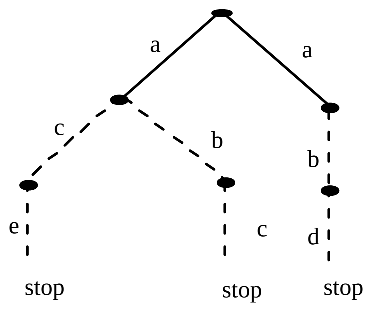


Figure 2.2: (a) : Projection of T1 (b) : Projection of T2

Assuming that the behavior of a reference specification can be represented by a tree structure, two basic strategies can be applied in order to find a possible execution history for the given trace: *depth-first* and *breadth-first* exploration. The difference between the two is in the order in which the set of possible branches are explored. In the depth-first strategy one branch is explored at a time, and a checkpoint is kept whenever a non-deterministic execution point occurs. Checkpoints are used to keep the context of execution and to restaure this context when the chosen alternative leads to a contradiction with the given trace, and another alternative must be explored. This restauration is called backtracking. It is used to cover, in a systematic manner, all possible paths in the reference specification. In the breadth-first strategy all the possible branches are explored in parallel. No checkpoints are needed, but it is clear that this strategy requires a large amount of memory space.

When specifications are of large size, performance of the search algorithm becomes very important. To establish a comparison, in terms of performance, between depth-first (DF) and breadth-first (BF) strategies, one has to consider different cases, distinguishing whether the given trace is valid or not, and considering different locations, of the validating branch, within the tree. Recursive branches leading to loops (see below) may also affect adversely the exploration performance. For invalid traces, both strategies are equivalent, at least as far as the number of branches to be explored, since all possibilities must be explored. The situation is the same when the given trace corresponds to the last branch in the behavior tree of the reference specification. In the other cases of valid traces, the DF strategy explores less branches. In Section 4.3.1, we compare the two discussed strategies through a concrete experience using the LAP-B specification.

In the case of recursive process behavior definitions, recursive calls may cause infinite loops during the trace analysis. The example below, for instance, may generate an infinite number of parallel processes of type P, where c is a hidden gate.

```
process P [ b] : noexit :=
   hide c in
              (c; P [b]
      |||
              b;  stop)
endproc
```

Having process P as a reference specification, the analysis of the trace "a;b" using a depth-first search in the behavior tree of process P, leads to an infinite loop as shown in Figure 2.3. Note that similar kinds of loops may also occur when the operators [], |[ ]|, or [> are used instead of ||| in the example above.

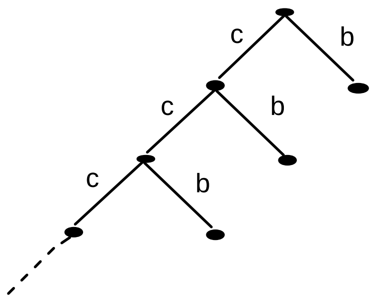


Figure 2.3: Infinite branches

The possible presence of such loops makes the problem of deciding the validity of a given trace with respect to a reference specification undecidable. However, we may arbitrarily limit the depth of exploration for internal actions to a specific limit. This will guarantee the termination of the analysis, but a negative result does not necessarily imply that the analyzed trace is invalid.

Our experience with real-life specifications of OSI communication protocols has shown that the number of branches produced during the exploration is often very large (see Section 4.3.2). Hence, the exploration of such behavior trees tends to be very slow regardless of the strategy used. A useful way to solve the problem would be to develop adequate heuristics that help to choose the branch most likely to be the one that corresponds to the analyzed trace.

Such heuristics could largely improve the analysis speed when the given trace is valid, which is the case in most instances, however, it would not speed up the analysis of an invalid trace, since all alternatives will be explored before a negative verdict is given. It seems that the LOTOS execution model described in [Wu 90] provides a suitable framework for exploring different exploration strategies, since it allows for arbitrarily controlled growing of the execution tree.

3. Design and implementation of a trace analysis tool: TETRA

This section describes the design and implementation of a trace analysis tool, called TETRA. This tool analyses an observed trace of interactions with respect to a reference specification which is assumed to be written in LOTOS [Loto 87]. The latter is a high level specification language, developed within International Organization for Standardization (ISO) for the development of formal specifications of the OSI communication protocols and services. This language has, however, a much larger scope of application. The language and the trace analysis tool TETRA follow the principles exposed in Section 2.

3.1. General description of TETRA

As shown in Figure 3.1, TETRA compares a given trace of interactions with a reference specification checking whether an execution history of the specification could produce the given trace. The result of the analysis is either "valid trace" or "invalid trace". In the latter case, an optional error diagnostic facility provides indications about possible causes of the discrepancy between the trace and the specification, as explained in Section 3.4.

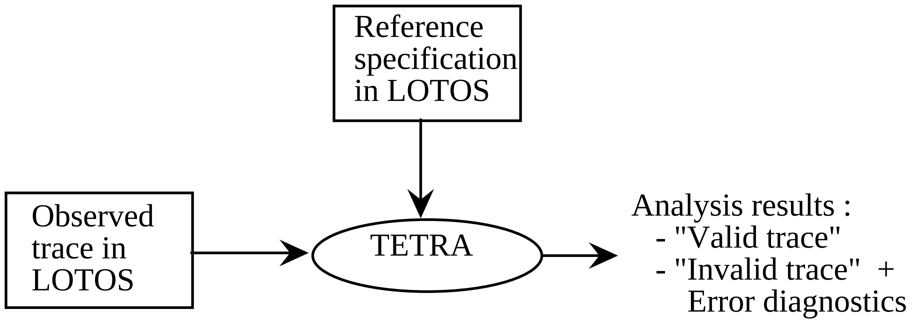


Figure 3.1: Trace analysis using TETRA

The first version of TETRA [Boch 89j] operates in batch mode where the reference specification is compiled together with the traces to be analyzed, which are written in the form of LOTOS processes. Subsequently, an on-line version of TETRA was developed [Saba 90] which compiles the reference specification alone and analyses a trace one interaction after the other. Although the present version of TETRA is not very efficient, the on-line version could be directly connected to a system under test and analyze the observed interactions as soon as they are executed. It detects an error as soon as possible, and the execution of the system under test could be halted in order to obtain further information about the system state at the time when the error occured.

The implementation of TETRA is based on the LOTOS interpreter ISLA [Logr 88] which operates in two phases. First the LOTOS specification is checked for syntax and typing errors and compiled into Prolog facts, and then the translated specification is interpreted by an interpreter written in Prolog. TETRA uses the ISLA compiler without change, while the LOTOS interpreter has been rewritten in order to perform the automatic trace analysis, as described in this section. The experiments reported in Section 4 were performed using TETRA written in Quintus Prolog running on a Sun SPARC station 1 workstation.

TETRA also has an option to validate the specification of test cases and their verdicts with respect to the reference specification which defines the expected behavior of the tested system. The application of this facility to the verification of standardized OSI conformance test cases is described in Section 4.2.

3.2. Interaction parameters

In the discussion of trace analysis in Section 2, we assumed that the interactions of the observed trace and the interactions defined in the reference specification have no data parameters. In most practical applications, however, data parameters of interactions must be considered. Also the distinction between input and output is important in this context. It is therefore important that a complete trace analysis tool analyses not only the order of the observed interactions in the trace, but also distinguishes whether the interactions were input to the system under test (or were produced as output) and whether the observed data values of output parameters are valid according to the specification.

TETRA provides the analysis of interaction parameters with respect to the information included in the reference specification written in LOTOS. In LOTOS, an interaction may have a number of parameters with well defined data types. The value of each parameter of an interaction is determined by all the processes that participate in the rendezvous interaction. A process that executes the behavior expression "g ! Value" is ready to participate in an interaction at the gate "g" provided that it has a single parameter with value "Value". A process that executes the behavior expression "g ? x:Type", where x is a variable, is ready to participate in an interaction if its parameter has a value of type "Type". The latter is a kind of "reception" which may be associated with a guard "g ? x:Type [<guard expression>]" which defines a boolean condition which must be satisfied for the interaction to be possible.

In the case of two communicating processes there are 3 possibilities concerning the use of the symbols "!" and "?":

- Value passing: one process using "!" determines the value which is accepted by the other process using "?".
- Value matching: Both processes use "!" and the interaction is only possible if both have selected the same value.
- Value generation: Both processes use "?" and an arbitrary value satisfying the conditions of both processes, if such a value exists, will be selected when the interaction is executed, and both processes know this value thereafter. If no such value exist, the interaction is not possible.

Data types, values and operations on data values are specified in LOTOS using a notation of abstract data types [Ehri 85]. These definitions are automatically evaluated by TETRA during the analysis of a trace. For this purpose, TETRA receives as input not only the name of an observed interaction (which is the same as the name of an external gate), but also the values of the associated parameters. TETRA accepts the parameter values in a notation which corresponds to the LOTOS source code for value expressions (which is also used by the interactive user interface of the ISLA interpreter). This notation is quite appropriate for abstract types, such as stacks and queues, however, the notation for "records" and basic predefined data types is very clumsy. In many practical applications therefore, the notation results in quite complex and unreadable type and value descriptions (see for example [Boch 89h] or [Boch 90j]).

Since the parameter values of the observed interaction trace usually are recorded by the test system in an implementation-dependent form, this form must be converted into the value notation accepted by TETRA prior to analysis. In the case of interoperability testing of implementations of OSI Application layer protocols, as discussed in Section 4.1, the observed interactions between the interworking OSI implementations are coded according to the ASN.1 encoding rules [ASN1 C], which are based on the data structures of the exchanged messages specified in the ASN.1 notation [ASN1]. Because of the regular structure of these coding rules, it is possible to use the ASN.1 definition of the standardized message structure to generate automatically the necessary coding routines that translate the ASN.1 encoded messages into the LOTOS notation accepted by TETRA, as explained in [Boch 89h].

3.3. Optimizations

Under this heading we discuss a few issues that are related to the efficient realization of trace analysis in the context of non-deterministic specifications. While the first subsection deals with a kind of state explosion which is related to the interleaving semantics for parallel activities, the other subsections deal with problems related to the data parameters.

3.3.1. Elimination of redundant branches

In addition to the invisible actions on hidden gates, as discussed in Section 2, LOTOS specifications may include other kinds of invisible events, such as the so-called internal event, written "i", which may be used to select an alternative, and the "exit" operation which terminates the behavior of a process. If such events occur within a behavior expression containing a parallel operator, they may cause the generation of several redundant branches, as shown for the expression "i; a; stop ||| b; c; stop" below, where all the dashed branches are redundant:

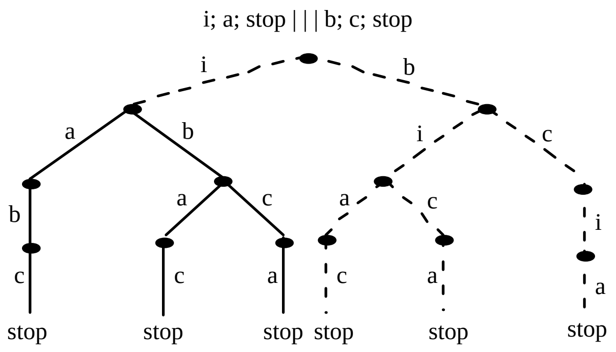


Figure 3.2: Redundant branches

In order to eliminate this redundancy, the following reduction rules might be applied :

a) i; B --> B

b) exit >> B --> B

c) exit (E_1, ..., E_n) >> accept x_1 : t_1, ..., x_n : t_n in B --> [E_1/x_1, ..., E_n/x_n] B

where E_i/x_i means that the value-identifier x_i in B is replaced by the value expression E_i.

The idea behind these rules is to eliminate any non-relevant internal event that would duplicate a number of branches in the behavior tree and increase the analysis time. After reduction, the resulting tree is trace equivalent with the original one, and so the analysis results will not be affected.

3.3.2. Uninstantiated parameters

A LOTOS specification may include a **choice** statement of the form "choice x:Type [<condition>] in <behavior expression>" which means that an arbitrary value of type "Type" satisfying the <condition> may be selected and then used during the execution of the subsequent behavior expression. Since the selected value may not have an impact on the first interactions of the behavior expression, but only on some interactions following later, the handling of this LOTOS feature in trace analysis is not so easy. It would be possible to assume one value for x and continue with the analysis until a contradiction between the trace and the reference specification is found. It would then be necessary to backtrack to the point of the **choice** statement and select another value, and so on, until all values have been tried. Since the set of possible values may not be finite, as in the case of Integers, this treatment of the **choice** statement may lead to infinite loops during the trace analysis.

We have therefore adopted another way of treating this case. Using the facility of Prolog of leaving variables uninstantiated until a value is found later on, we represent the selected value of a **choice** statement by a Prolog variable which remains non-instantiated until its value can be deduced from the observed output interactions contained in the subsequent trace.

The same problem occurs when an interaction occurs on an internal (invisible) gate of the reference specification and for one of the parameters the value must be generated because none of the processes participating in the interaction defines its value (see Section 3.2, case of value generation). TETRA treats such a parameter in the same manner as a **choice** variable.

3.3.3. Elimination of unfeasible branches

During the expansion of the behavior tree of the reference specification, many of the explored branches contain predicates which should be true. In constraint-oriented specifications [Viss 88], the predicate of a branch is often the conjunction of several independent conditions imposed by different processes participating in an interaction, and sometimes this different conditions contradict one another. We call a branch "unfeasible" if its attribute evaluates to False.

Unfortunately, often these predicates contain variables which remain uninstantiated, as explained above, and they cannot be evaluated as long as they contain an uninstantiated variable. In general, the question whether a given branch is unfeasible, that is, whether its predicate evaluates to true, is undecidable. In order not to loose a possibility of explaining the observed trace as a valid execution history, TETRA assumes that a predicate with uninstantiated variable will evaluate to True. In addition, the user may establish a database of contradictory predicates which is consulted by TETRA to detect unfeasible branches. This approach has been used successfully in the case of trace analysis with respect to the OSI Transport service [Saba 90].

3.4. Error diagnostics

While the basic function of a trace analysis tool is the determination of the validity of the analyzed trace, it would clearly be useful to also provide some diagnostic facilities which would identify the reason for non-conformance in the case of an invalid trace. The provision of a function which provides meaningful and intuitively "correct" diagnostics is very difficult. The problem is similar to, if not more difficult than, the problem of providing meaningful error diagnostics in compilers.

TETRA includes a second "diagnostic" phase which is initiated only for non-conforming traces. During this phase various error hypothesis are checked for consistency with the analyzed trace. Each hypothesis gives rise to a diagnostic message, which may be of the form "The second interaction is wrong and should be such and such", or "The third interaction of the trace should be absent". Each diagnostic message gives a possible interpretation of the reason for the non-conformance of the analyzed trace.

A first type of error hypothesis is called "wrong actions". It includes the following cases:

(1) Gate error: a different gate should be used for a given interaction.

(2) Parameter error: either the number and/or types of the parameters of a given interaction are not correct, or a parameter value should be different.

(3) Predicate error: in the case that a guard is associated with the interaction, the predicate of the guard evaluates to false in the reference specification.

TETRA performs a depth-first search of the behavior tree of the reference specification and finds all possibilities of N action errors that are sufficient to explain the analyzed trace with respect to the specification. N is a parameter which can be set by the user. It usually takes the value 1 or 2. If for a given branch of the specification behavior tree one needs more than N errors to explain the analyzed trace, the specification branch is abandoned. If the analysis reaches the end of the trace, a diagnostic is obtained corresponding to the errors that must be assumed in order to equate the analyzed branch with the given trace. Several different branches may therefore lead to different diagnostics.

As an example, we consider the reference specification Example 3.1 (which is similar to Example 2.1) and the non-valid trace T="a;e;d". The analyzer will first explore the first alternative of the specification (line 7) and encounter a discrepancy with the specification at the second interaction. In the case that the maximum number N of errors to be considered is one, this alternative will not be further explored since the following interaction of the trace cannot be explained. The same happens to the second branch (lines 7 and 9). Therefore the analyzer will backtrack and explore the third alternative of the specification (line 11). It finds that the trace can be explained with one wrong action at the second place, which is indicated by the diagnostic message "action 2 should be: b!x:Nat [x ne 0]". In the case of N equal to 2, the first alternative would also give rise to a diagnostic message, which would be of the form "action 2 should be: c!0 ; action 3 should be: e".

In this example, it seems intuitively clear that the error in trace T is caused by the second interaction. However, in more complicated situations, it is not easy to identify the cause of the error. In any case, the analyzer provides a number of possible diagnostic messages, each indicating how the analyzed trace could be obtained from a correct branch by a small number of changes.

Example 3.1

```
(1)  specification   S [a,b,c,d,e] : noexit :=
(2)  Library   NaturalNumber   endlib
(3)  behavior
(4)          P[a,b,c,d,e]
(5)  where
(6)          process  P[a,b,c,d,e] : noexit :=
(7)                  a; (c!0; e; stop
(8)                       []
(9)                       b; c; e; stop)
(10)         []
(11)         a; b ?x:Nat [x ne 0]; d; stop
(12) endproc
     endspec
```

A second type of error hypothesis concerns additional and missing actions. TETRA checks whether the trace includes an interaction that should not be present according to the specification, or whether an interaction foreseen by the specification is not present in the trace. The analysis proceeds in a similar manner as explained for the case of wrong actions. Again, a user-chosen parameter N limits the number of errors of this type that are considered by the analyzer.

The following table shows further examples. Various traces are compared with the specification Example 3.1 and the diagnostics obtained are indicated. The value N=1 is assumed, except for the trace marked by a "*" which requires N=2 for providing any diagnostic. The last three diagnostics show that different error hypothesis may explain a given trace with respect to the same branch in the specification.

Trace	Diagnostic(s)	Line # in the spec.	Error type
a; c!0; d	action 3 should be: e	7	erroneous action gate
	action 2 should be: b!x:Nat [x ne 0]	11	erroneous gate and param.
a; c!Succ(0); e	action 2 should be: c!0	7	erroneous action parameter
a; c; e	action 2 should be: c!0	7	missing parameter
a; b!0; d	action 2 should be: b!x:Nat [x ne 0]	11	predicate evaluated to false
b; c; e *	action 1 should be: a and action 2 should be: c!0	7	erroneous action gate & missing parameter
a; c!0; c!0; e	action 3 should be absent	7	extra action
a; d	action b ?x:Nat [x ne 0] is missing, should be in position 2	11	missing action
a; b; e	action 2 should be: c!0	7	erroneous action gate & missing parameter
	action 3 should be: c	7 & 9	erroneous action gate
	action 3 should be absent	7 & 9	extra action
	action c is missing, should be in position 3	7 & 9	missing action

4. Experience with the TETRA tool

4.1. Experience with test result analysis

We consider in this subsection the analysis of test results during the testing of an IUT. In the case of standardized test cases including verdicts, the analysis of the test results can be performed based on the verdicts, however, such an approach is not possible when other test cases are used which may be required for additional test coverage, the testing of implementation-dependent features or interoperability tests. In all such cases, including the case of random test inputs, the test results can be analyzed directly with respect to the specification [Boch 89m] using automated tools such as TETRA.

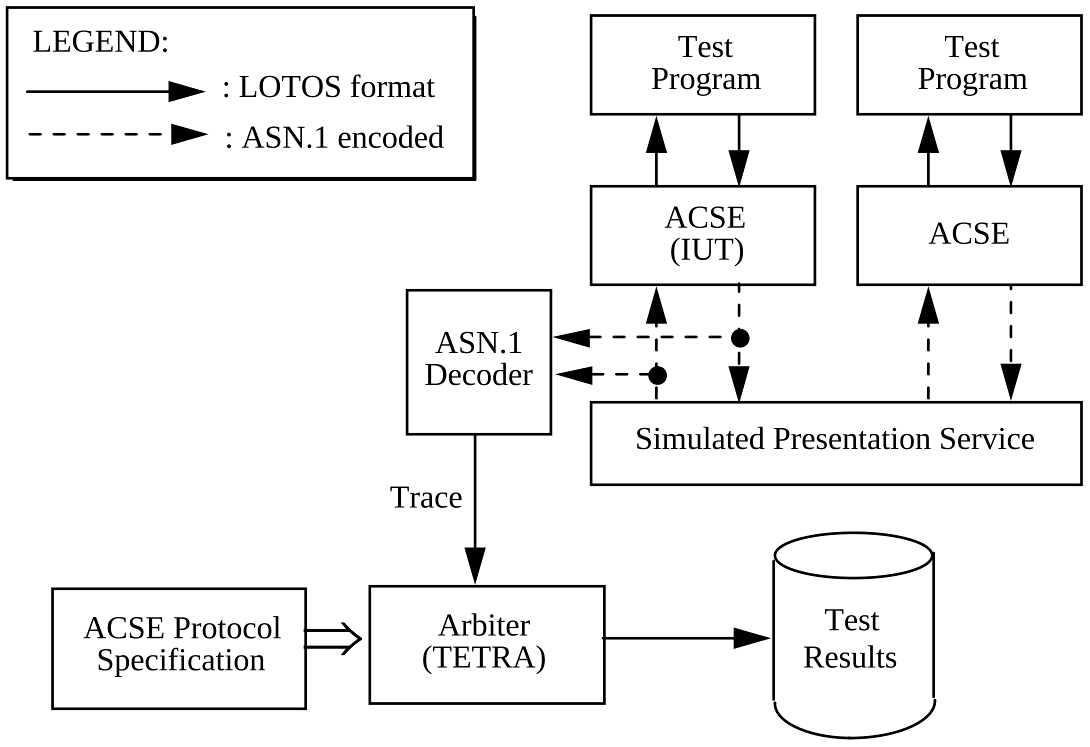


Figure 4.1: ACSE implementation test experiment

An experiment with test result analysis has been performed for the protocol of the Association Control Service Element (ACSE), which is an OSI Application layer protocol. The purpose of this experiment was at the same time to demonstrate tools for ASN.1 which were developed for implementation support in conjunction with the FDT Estelle [Boch 90f], and for the support of ASN.1 in relation with LOTOS [Boch 89h]. The ACSE protocol was used in this experience because of its relative simplicity. The experiment consisted of having two ACSE implementations communicate with one another and having the exchanged PDU's observed and automatically analyzed by the trace analysis tool TETRA, as shown in Figure 4.1. These implementations, written in C, were obtained through the automatic translation of an ACSE Estelle specification. The latter contained PDU definitions automatically obtained through translation from the original ASN.1 definitions found in the standard [ISO 8650]. The implementation contained automatically generated PDU encoding and decoding routines.

The exchanged PDU's were recorded into a trace file and at the same time analyzed on-line by the TETRA tool using as the reference an ACSE LOTOS specification. The latter also contained PDU definitions automatically obtained through translation from the ASN.1 definitions. Before being analyzed by the TETRA tool, which accepts the analyzed interactions in LOTOS action format, the ASN.1-encoded PDU's were translated into the form of LOTOS expressions by the ASN.1 Decoder (see Figure 4.1) automatically generated by our ASN.1/LOTOS tool.

The ACSE protocol specification is relatively simple. Nevertheless, the length of the PDU definitions in ASN.1 is 100 lines. This is translated into 200 lines of Estelle type definitions and 1000 lines of LOTOS data type definitions. The total size of the ACSE LOTOS specification used by TETRA as a reference is 2400 lines. The control part is small (approximatively 200 lines) which gives a simple behavior tree. The on-line version of TETRA validates each interaction in matters of seconds. A certain number of test scenarios were run and the resulting traces of PDU's were analyzed. TETRA detected one error in the implementation and one error in the specification. In all those cases, the diagnostic part located the erroneous behaviors. Example 4.1 shows an execution trace of TETRA which points out an error in the implementation.

Example 4.1: Execution trace of TETRA (on-line version)

Observed new interaction ->

P !Input:IO !PCONind : primitive !ACSE_apdu(AARQ_apdu(**protocol_version(Bit(0)+Bit(1))**, application_context_name(a), called_AP_title(d), called_AE_qualifier(e), called_AP_invocation_id(Not_Present), called_AE_invocation_id(Not_Present), calling_AP_title(b), calling_AE_qualifier(c),calling_AP_invocation_id(Not_Present), calling_AE_invocation_id(Not_Present),, implementation_information(Not_Present), user_information(f+ <>))) : ACSE_apdu

... Valid ...

Observed new interaction ->

P !Out : IO !PCONrspAcceptance : primitive !ACSE_apdu(AARE_apdu(**protocol_version(Bit(1))**, application_context_name(a), **result(0)**, Associate_source_diagnostic(acse_service_user(0)), responding_AP_title(Not_Present), responding_AE_qualifier(Not_Present), responding_AP_invocation_id(Not_Present), responding_AP_invocation_id(Not_Present), implementation_information(Not_Present), user_information(Not_Present)))) : ACSE_apdu

... Invalid ...

Next Possible Actions

<1>- P !Out:IO !PCONrspUserRejection:primitive !ACSE_apdu(ACSE_apdu_genere_1(AARE_apdu(protocol_version(**protocol_version(Bit(1))**, application_context_name(a), **result(Succ(0))**, Associate_source_diagnostic(acse_service_provider(Succ(Succ(0)))), responding_AP_title(Not_Present), responding_AE_qualifier(Not_Present), responding_AP_invocation_id(Not_Present), responding_AP_invocation_id(Not_Present), implementation_information(Not_Present), user_information(Not_Present))) : ACSE_apdu

<2>- P !Out:IO !PUABreq : primitive !ACSE_apdu(ABRT_apdu(abort_source(Succ(0)), user_information(Not_Present))) : ACSE_apdu

<3>- ...

In this example, we wanted to test if the called implementation reacts properly when an AARQ APDU is sent with an unsupported protocol version (first interaction of Example 4.1). In this test scenario, the called implementation should respond with a rejection, i.e. AARE APDU (Result = 1), but instead, it responds with an acceptance, i.e. AARE APDU (Result = 0), as shown in the second interaction of Example 4.1. In a first analysis phase, TETRA diagnoses an invalid interaction. In a second phase, TETRA provides a list of actions that could have validly taken place instead of the erroneous behavior. This includes the AARE APDU (Result = 1) shown as last interaction in Example 4.1 (note that "Succ(0)" is the notation for 1 in LOTOS).

One of our test cases (collision of two release requests) highlights a problem with remote and distributed testing architectures, which may also apply to local testing when the interfaces contain queues. According to the specification, the IUT may generate a RLRQ (release request) PDU just before receiving a RLRQ from its peer, as shown for the initiator in Figure 4.2, but not after it has received such a PDU. However, the latter sequence may be observed by an observer that resides at the peer site, or somewhere between the two communicating entities, as shown by the dashed line in Figure 4.2.

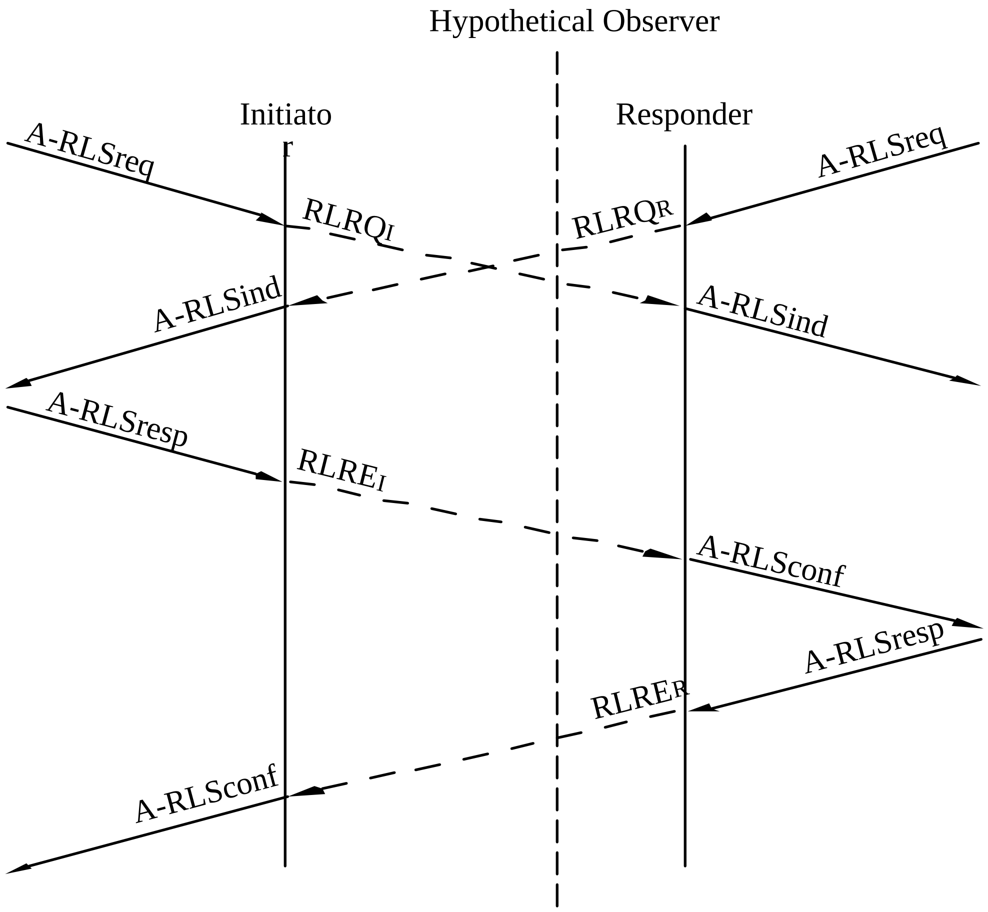


Figure 4.2: Sequence of ASPs and APDUs in the release collision scenario

Although we performed local observation, as shown in Figure 4.2, we sometimes observed the wrong sequence of PDU's. This is due to the fact that our point of control and observation (PCO) was at the Presentation service interface, and included queues for PDU buffering within the ACSE entity implementation. It seems that this is a quite normal situation. Unfortunately, this makes the observation of certain timing and ordering errors, such as the one above, difficult to observe and diagnose [Dsso 90].

The presence of such queues between the PCO and the state machine of the protocol implementation can be taken into account during the test result analysis by including queues in the reference specification used for the analysis, as shown for X.25 in Figure 4.3. We did not do this exercise in the case of ACSE.

4.2. Experience with test case validation

The test cases developed for OSI conformance testing usually contain verdicts for different possible reactions of the IUT. The verdict has usually the form of "test passed", "test failed", or "inconclusive". The "inconclusive" verdict indicates that the observed behavior of the IUT satisfies the rules of the specification, but that the particular behavior to be tested could not be observed. Since it is important that these verdicts conform to the protocol specification, and test descriptions are usually quite lengthy and complicated, it would be useful to automatically check the verdicts of a defined test case against the specification of the protocol. As long as the test case consists of a finite number of possible traces (each ending with a particular verdict), the verification of the test case is in fact possible by performing the same kind of test trace analysis, as described above, separately for each trace of the test case.

The main problem for the automation of such a test case verification is the availability of a suitable formal specification of the protocol specification, to be taken as the reference. In addition, the test case should be defined in a compatible language. Most OSI test cases are now written in the TTCN language [OSI C3]. It seems that for the comparison of a test case with a formal protocol specification, it is most convenient to represent the test case in the same language in which the specification is written. Therefore the automatic translation of TTCN into FDT's would be a useful step towards the validation of test cases (see for instance [Sari 88f]).

We have validated the verdicts of a large number of the ISO/CCITT conformance test cases for the link layer of X.25 [ISO 8882] against a specification of the corresponding protocol, the LAP-B, as mentioned in the introduction. For this purpose, we used an existing LOTOS specification of LAP-B which had already been validated through extensive simulations [Guer 89a]. This is a quite sizable specification of approximately 2500 lines of LOTOS code. Since the test suite was described in TTCN, we had to translate it into LOTOS. This has been done manually [Dubu 90].

As shown in Figure 4.3, the test cases are executed under a simulated remote test architecture. We chose this architecture because the test cases of the ISO document were conceived to be executed within such an architecture. The test cases only describe the interactions with the lower tester. The medium is a reliable full duplex queue.

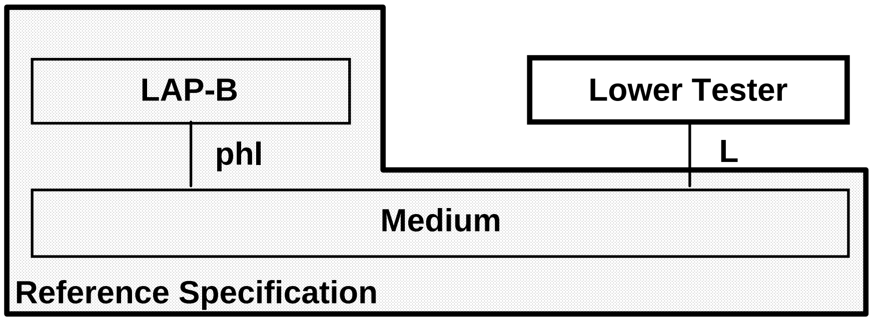


Figure 4.3: Remote Test Architecture

The complexity of the behavior tree increases when we model the medium between the specification and the lower tester explicitly, as part of the reference specification, in the form of two FIFO queues in LOTOS. For simple test cases, the analysis time increases by a factor of up to ten. In more elaborated test cases, the analysis aborts due to a lack of memory. In order to obtain our results, we have therefore bypassed the queues. Table 4.1 shows results for the validation time of some of the test cases. The following paragraphs discuss particular aspects of our results.

Test case	Number of branches isolated by TETRA	Number of branches validated	Maximum number of test steps in a branch	Total validation time (in sec.)
DL1-101	21	21	7	1541
DL1-207	18	18	7	974
DL2-101	98	8	7	3005

Table 4.1: Statistics on validation time

(a) Detected error in LAP-B test suite: We have found an error in one of the test cases of the original TTCN document. Test case DL1_306 says that the trace:

```
L ! DISC (P:=1)
  L ? DM [F=1]
    L ! UA (F:=1)
      L ? DISC [P=1]
```

should have a fail verdict, but it is accepted by the specification. We found that this sequence of actions is valid with respect to the LAP-B standard. We believe that the last test step of the test case DL1_306 (L ?Otherwise) should have an inconclusive verdict instead of a fail verdict.

(b) Detected error in the specification: Test case DL1_207 indicates an error in the specification which does not include all details concerning error processing. The branch:

```
L ! DISC (P:=1)
  L ? DM [F=1]
    L ! Hex (string:='03F??'H)
      L ? Otherwise
```

has a fail verdict, but is accepted by the specification (string '03F??'H is a SABM/P=1 with an non-empty information field).

(c) Preambles for arbitrary initial states: We were surprised to see that TETRA rejected certain branches of preambles which are considered valid according to the test suite [ISO 8882]. For instance the branch:

```
L ! DISC [P:=1]
  L ? UA [F=1]
```

of the subtree DL1_STATE is not accepted by the LAP-B specification. Later we noticed that the ISO test cases do not necessarily assume that the IUT is initially in the disconnected state. For instance, the above branch is valid starting in the data transfer phase.

(d) Error diagnostics: While this facility works well for smaller specifications, we found that in this example the number of fault hypothesis (each indicated by a diagnostic message) was often too large to be useful. For instance, a fault in the first actions of a branch could lead to about hundred diagnostics messages.

4.3. Performance considerations

During the experience with the LAP-B test case validation described above, we were interested in determining the performance of the analyzer when different search strategies are used, and in understanding how the behavior tree of the reference specification changes during the validation of a typical test case. This latter aspect concerns the branching factor in the specification behavior tree, and the kinds of branches generated at each step of the validation process.

4.3.1. Space and time efficiency of different search strategies

Two search strategies have been implemented in TETRA for the sake of comparison, namely, *Breadth-First (BF) strategy* and *Depth-first (DF) strategy*. We used these two strategies to validate the test case DL2-101. This test case contains 36 branches with a maximum of five actions per branch. Table 4.2 summarizes the validation time, in seconds, taken by TETRA to validate each branch. Note that in the case of the BF strategy, the analyzer could not validate all the branches because of memory limits. We conclude from Table 4.2 that the DF strategy is a good factor more efficient than the BF strategy.

Table 4.2

Branch	Depth-First Search	Breadth-First Search
1	108.7	170.72
2	10.3	155.35
3	17.27	54.37
4	8.0	49.41
5	13.87	39.88
6	2.55	38.38
7	2.15	38.33
8	6.8	37.45
9	6.72	38.25
10	13.47	20.65
11	13.25	20.82
12	13.27	20.45
13	13.22	20.70
14	13.17	21.47
15	87.57	175.67
16	6.95	21.98
17	6.95	22.1
18	7.03	30.58
19	6.93	35.5
20	6.93	27.77
21	7.03	34.37
22	6.88	26.68
23	531.3	system overflow
...	...	

4.3.2. Evolution of the reference specification behavior tree

In order to better understand the evolution of the behavior tree of the reference specification during the analysis of a trace, we have obtained some statistical information about the number of possible branches compared with the total number of branches foreseen by the specification, and the relative number of internal actions compared with external ones. This information was obtained using the breadth-first search strategy.

In order to illustrate the kind of information which was collected, Figure 4.4 shows a very simple example of a behavior tree which was explored during the analysis of a given trace. The dashed arrows correspond to branches which are not further explored since the behavior defined for this arrow is in contradiction with the analyzed trace. The thin arrows correspond to internal actions, which are not externally observable, and the fat arrows correspond to an interaction defined by the specification which is matched by the observed trace.

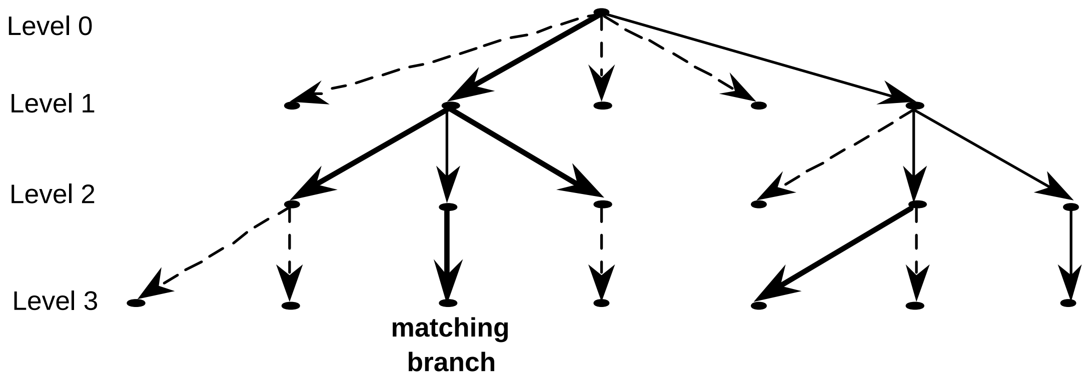


Figure 4.4: Reference behavior tree evolution, illustration

Table 4.3 shows the information which would be gathered for the above example. For each level of the behavior tree (throughout the whole breadth), the first two columns indicate the total number of branches, and the number of feasible ones. The last two columns correspond to the number of thin and fat arrows. The redundant branches indicated in the third column are branches that have the same action (internal or external) as another branch at this level, and also the same behavior expression, which determines the subsequent behavior. Because they are redundant, these branches need not be further explored.

Table 4.3

Level	Total number of branches	Number of possible branches			
		total	redundant	with internal action	with external action
1	5	2	0	1	1
2	6	5	0	3	2
3	7	3	0	1	2
Number of actions on matching branch: 1 internal, 2 external. Analysis time : n seconds					

Tables 4.4 and 4.5 show the same kinds of results for the analysis of two typical traces, corresponding to two branches (number 15 and 23, respectively) of the LAP-B test case DL2_101 already mentioned in Section 4.2. We can draw the following conclusions from these experiments:

(a) Number of branches with internal actions: For the LAP-B reference specification, the number of branches beginning with internal actions is always clearly greater than the number of branches beginning with external actions. However, specifications written in a purely constraint-oriented style, such as the OSI Transport service specification [OSI TS3] used in [Saba 90], have no internal actions. In this latter case, only a single fat arrow was observed at each level, together with a large number of unfeasible branches.

(b) For the matching branch, the number of internal actions is small compared with the number of external actions observed (at least in the case of this LAP-B specification).

(c) The analysis time required for the analysis of a trace varies a lot, even if the traces have the same length. In the case of the LAP-B specification, this fact can be explained by the blow-up of alternative branches which is observed at certain levels of the analysis, such as at the deeper levels in the Tables 4.4 and 4.5.

Table 4.4

Level	Total number of branches	Number of possible branches

		total	redundant	with internal action	with external action
1	11	9	7	8	1
2	19	17	8	16	1
3	44	28	18	20	8
4	28	24	17	22	2
5	34	24	17	19	5
6	28	24	16	22	2
7	24	12	6	6	6
8	2	2	0	2	0
9	8	4	0	2	2
10	10	6	2	4	2
11	74	70	2	68	2
12	152	148	2	82	66

Number of actions on matching branch: 7 internal, 5 external.
Analysis time : 240.666 seconds

Table 4.5

Level	Total number of branches	Number of possible branches			
		total	redundant	with internal action	with external action
1	11	9	7	8	1
2	14	12	8	11	1
3	12	6	3	3	3
4	1	1	0	1	0
5	4	2	0	1	1
6	5	3	1	2	1
7	37	35	1	34	1
8	140	74	1	41	33
9	467	387	252	347	40
10	628	438	273	343	95
11	4076	3936	3593	?	?
12	Not	enough	memory		

5. Conclusions

We have shown in this paper that the specification of a tested system may be used as a reference for the automatic analysis of test results, not only in the case of deterministic specifications where the output interactions and their parameters are determined from the given test inputs, but also in the case of non-deterministic specifications which allow for different outputs for a given sequence of inputs. However, in the latter case, test result analysis is more complex since different execution histories of the specification must be explored in order to check whether one of them may explain the outputs received from the system under test. Although this question is non-decidable in general, we explain in this paper how such an analysis can be performed in many practical situations.

We have considered, in particular, the case that the reference specification is written in LOTOS, which is a formal specification language developed by ISO for the description of OSI communication protocols and services. A trace analysis tool, called TETRA, is presented, which analyses a given trace of interactions with respect to a reference specification written in LOTOS. Our practical experiments, mentioned in Section 4, indicate that TETRA is already a practical tool for the analysis of test results and the validation of verdicts in conformance test cases. We hope that further optimizations of the analysis and diagnostic algorithm will lead to an improved tool which can handle most practical problems in this area. However, our implementation approach using Prolog is probably not sufficiently efficient for most cases of on-line analysis of test results.

The performance studies discussed in the paper indicate, as could be expected, that a depth-first strategy for the exploration of all possible behaviors of a reference specification is more efficient than a breadth-first exploration. However, a breadth-first exploration provides interesting statistics about the structure of the behavior tree in terms of number of possible branches, and internal actions. This structure largely determines the effectiveness of the trace analysis algorithm and it is dependent on the structure of the reference specification. Further work is required to better understand the impact of the specification style on the ease of trace analysis.

The errors found during our experiences indicate, as could be expected, that even well studied specifications, implementations and test cases still contain a few errors. In particular, it shows that the automatic checking of OSI conformance test cases with respect to the corresponding protocol specification is a useful activity for increasing the confidence in the OSI specifications. It is important to note that the automation of this activity is only possible when a formal specification of the protocol is available. Unfortunately, at present, there are only few formal specifications of OSI protocols or services that have been generally recognized to faithfully represent the OSI standards.

References

[ASN1] ISO 8824 (1987) *Specification of Abstract Syntax Notation One (ASN.1).*

[ASN1 C] ISO 8825, *Information Processing - Open systems Interconnection - Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).*

[Boch 89h] G. v. Bochmann and M. Deslauriers, *Combining ASN.1 support with the LOTOS language*, Proc. IFIP Symp. on Protocol Specification, Testing and Verification IX, North Holland Publ., 1989, pp. 175-186.

[Boch 89j] G. v. Bochmann and O. Bellal, *Test result analysis with respect to formal specifications*, Proc. 2nd Int. Workshop on Protocol Test Systems, Berlin, Oct. 1989.

[Boch 89m] G. v. Bochmann, R. Dssouli and J. R. Zhao, *Trace analysis for conformance and arbitration testing*, IEEE Transaction on Software Engineering, Nov. 1989, pp.1347-1356.

[Boch 90g] G. v. Bochmann, *Protocol specification for OSI*, Computer Networks and ISDN Systems 18 (April 1990), pp.167-184.

[Boch 90j] G. v. Bochmann, D. Desbiens, M. Dubuc, D. Ouimet and F. Saba, *Test result analysis and validation of test verdicts*, Workshop on Protocol Test Systems, Washington, Oct. 90.

[Boch 91d] G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc and A. Ghedamsi, *Fault models in testing*, Proc. IFIP Intern. Workshop on Protocol Test Systems, Netherlands, Oct. 1991 (invited paper).

[Bolo 87] T. Bolognesi and E. Brinksma, *Introduction to the ISO Specification Language LOTOS*, Computer Networks and ISDN Systems, Vol. 14, No. 1, pp. 25-59, 1987.

[Cork 83] R. Cork, *The testing of protocols in SNA products - an overview*, Proceedings of the IFIP WG 6.1 Third International Workshop on Protocol Specification, Testing, and Verification, pp. 455-463, June 1983.

[Dubu 90] M. Dubuc, G. v. Bochmann, O. Bellal and F. Saba, *Translation from TTCN to LOTOS and the validation of test cases*, Proceedings FORTE '90 (IFIP), 1990.

[Ehri 85] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 1*, Springer Verlag, 1985.

[Guer 89a] D. Gueraichi and L. Logrippo, *Derivation of Test Cases for LAP-B from a LOTOS Specification*. In: Vuong, S.T. (Ed.) Formal Description Techniques II. North-Holland, 1990, pp. 361-374.

[ISO 8650] ISO 8650 (1988) *Protocol Specification for the Association Control Service Element (ACSE).*

[ISO 8882] ISO, *X.25-DTE Data link layer conformance test suite*, TC97/SC6, DIS 8802 Part 2, 1989.

[Jard 83b] C. Jard and G. v. Bochmann, *An approach to testing specifications*, Journal of Systems and Software, Vol.3, 4(Dec. 1983), pp.315-323.

[Logr 88] L. Logrippo and al., *An interpreter for LOTOS: A specification language for distributed systems*, Software Practice and Experience, Vol. 18 (4), pp. 365-385, April 1988.

[Loto 89] ISO, IS8807 (1989), *LOTOS: a formal description technique*.

[Matt 88] R. Matthews, K. Muralidhar, and S. Sparks, *MAP 2.1 conformance testing tools*, IEEE Transactions on Software Engineering, vol. 14, no. 3, pp. 363-374, March 1988.

[Molv 85] R. Molva, M. Diaz, and J. Ayache, *Observer: A run-time checking tool for local area networks*, Proceedings of the IFIP WG 6.1 Fifth International Workshop on Protocol Specification, Testing, and Verification, pp. 495-506, June 1985.

[OSI C3] ISO, DP 9646-3, *Information Processing Systems - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation (TTCN).*

[OSI TS3] ISO TC97/6/WG4/N317, *Formal Description of ISO 8072 in LOTOS,* 1987.

[Prob 88] R. Probert, *Towards a knowledge-based model for conformance test results analysis*, Proceedings of the First International Workshop on Protocol Test Systems, October 1988.

[Rayn 87] D. Rayner, *Standardizing Conformance Testing for OSI*, Computer Network and ISDN Systems, 14 no.1 (1987), pp. 79-98.

[Saba 90] F. Saba, *Validation en ligne de traces d'exécution appliquée au protocole de Transport*, M.Sc. thesis, Université de Montréal, fall 1990.

[Sari 88f] B. Sarikaya, *Translation of test specifications in TTCN to LOTOS*, Proc. IFIP Symposium on Protocol Specification, Testing and Verification, Atlantic City, 1988.

[Sari 89c] B. Sarikaya, *Conformance Testing: Architectures and Test Sequences*, Computer Networks and ISDN Systems 17, 1989, pp. 111-126.

[Ural 86] H. Ural and R. Probert, *Step-wise validation of communication protocols and services*, Computer Networks and ISDN Systems, vol. 11, no. 3, March 1986, pp. 183-202.

[Viss 88] C. Vissers, G. Scollo and M. v. Sinderen, *Architecture and Specification Style in Formal Descriptions of Distributed Systems*, Proc. IFIP Symposium on Prot. Spec., Verif. and Testing, Atlantic City, 1988.

[Wu 90] C. Wu and G. v. Bochmann, *An Execution Model for LOTOS Specifications*, GLOBCOM '90, San Diego, Dec. 1990.