TEST RESULT ANALYSIS WITH RESPECT TO FORMAL SPECIFICATIONS

Gregor v. BOCHMANN and Omar B. BELLAL

Université de Montréal
Montréal, Canada

Abstract:

There are two aspects to testing: (1) the selection of appropriate test inputs and (2) the analysis of the observed interactions of the implementation under test (IUT) in order to determine whether they conform to the IUT's specification. The paper considers the second aspect with particular attention to the testing of OSI communication protocol implementations. A system is described which analyses an observed test trace of interactions with respect to a reference specification which is assumed to be written in LOTOS. In the case that an error is detected, the system also provides some diagnostic information for locating the "error" in the analyzed trace. The practical use of such a trace analysis system is discussed, as well as the possibility of using a similar approach for the validation of the verdicts which are included in the standardized OSI conformance test cases.

1. Introduction

1.1. Need for automatic test result analysis

One difficult problem in system testing is usually 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 conform with the specification of the IUT. This paper deals with the construction of such oracles, and their use in the protocol testing process.

For established OSI protocol standards, such as X.25, FTAM or MHS (X.400), a number of standard test cases are defined by interested groups and/or standardization bodies. These test cases not only include the inputs to be applied to the IUT, but also describe the possible outputs observed, and for each possible output whether its occurrence means a successful test, the detection of an error, or an inconclusive test outcome. If such test cases are applied to an IUT, it is not necessary to have an oracle, since the test case definition already includes the information provided by the oracle. However, there are cases where these standard test cases cannot be applied, as explained below.

(a) During conformance testing of an implementation, it is often desirable to execute tests which have not been foreseen by the implementor. New test cases may be selected for this purpose. A similar situation occurs in hardware testing where test cases are sometimes selected randomly.

(b) While standard OSI conformance test cases are defined for verifying conformance of an implementation to the protocol specification, each implementation usually has to satisfy additional requirements which are system-specific. Specific test cases for verifying these requirements must therefore be designed and executed.

(c) Another case where the standard OSI conformance test cases cannot be used is arbitration testing. Arbitration testing is performed when two implementations, already well tested, do not interwork properly. In order to determine the reason for the problem arbitration testing is usually performed in a configuration which includes a module which observes the interactions between the two implementations under test, which communicate with one another, as shown in Figure 2. The observer module in Figure 2 must include an oracle function in order to be able to decide whether any protocol implementation behaves in contradiction to the protocol specification.

(d) Finally, the standard test cases are usually not used for the initial debugging of a new implementation. In the early implementation phase, it is often desirable to be able to executed ad hoc test cases and verify the behavior or the implementation in these situations. Similarly, specific tests may be selected during conformance resolution testing [OSI C] to check particular conformance requirements.

This paper takes the view that the selection of test cases should be separated from the problem of deciding whether the IUT behaves according to the specification for a particular test case. This latter problem can be handled by automatic analysis of the trace of observed interactions to a reference specification.

1.2. Test selection versus trace analysis

Figure 1 shows the distributed test architecture for the testing of protocol implementations, one of the standardized architectures for OSI conformance testing [OSI C]. It shows the Upper and Lower Test modules which observe the interactions at the upper and lower interfaces of the IUT, respectively. They also determine the test input provided to these interfaces during test execution.

We consider in this paper a testing approach where the concern for selecting the appropriate test input provided to the IUT is separated as much as possible from the analysis of the observed output [Dsso 85, Boch 89k]. We therefore consider separately:

(1) the selection of the test cases, and

(2) test result analysis: the determination whether the observed output trace conforms to the specification of the IUT.

The first aspect is important since the applied test input determines to a large extent what kind of IUT malfunctions can be detected. We assume in general that the second function is performed in a manner independent of the specific test input applied.

While the validity of an output usually depends on previous test input, we assume here that the test trace analysis is performed based on the IUT specification, independent of the test case to be executed.

1.3. Validation of the verdict of test cases

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". 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 below, 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]).

1.4. Overview of this paper

The error detection power of test trace analysis depends on the test architecture. Various architectures are discussed in Section 2 in relation to test result analysis. Section 3 describes the problem of test trace analysis with respect to protocol specifications written in LOTOS. A particular analysis system, based on an existing LOTOS interpreter [Logr 88], is also presented. In addition to deciding whether a given trace is valid to a reference specification, this system also provides some error diagnostics which are discussed in Section 4. Section 5 gives some conclusions.

2. Test architecture and observability

2.1. Error detection power

It is important to note that in a distributed test architecture as shown in Figure 1, each local tester can only detect certain errors. The lower tester in Figure 1, for instance, does not see the service primitives executed by the IUT, and therefore cannot detect any error related to the service interactions, unless it contains certain knowledge about the service interactions applied as test input by the upper tester. A more detailed discussion of the error detection power of various test architectures is given in [Dsso 86, Dsso 86b, Boch 89k].

In the following we assume that the trace analysis function is intended for use with arbitrary test cases. It should therefore not rely on any particular features of a test case. A trace analysis module for the lower tester in Figure 1, for instance, cannot make any assumptions about the service primitives and their parameters executed by the IUT at its upper interface. This limits its error detection power. For instance, such a trace analysis module cannot verify that user data sent by the upper tester is correctly transferred. However, it can verify that all rules concerning the exchange of PDU's between the IUT and its peer are correctly followed.

In most testing architectures, the error detection power of the lower tester (see Figure 1) is further decreased due to the fact that the "lower" interface of the IUT are not observed directly, but only through an underlying communication service. The latter usually implies delays and possible queuing of messages in transit. Even if the communication service has the FIFO property (in a given direction, messages are received in the same order as they were sent), the order of interactions observed may be different from the order they occurred as the IUT interface due to message cross-over within the communication service.

We call "reference specification" the set of rules which are verified for each analyzed test trace. As discussed in [Dsso 86b, Boch 89k], the reference specification for a particular test architecture can be obtained from the specification of the IUT by making abstraction from the interactions which are not observed (e.g. certain service primitives) and possibly performing a composition with the specification of the medium through which the IUT is observed.

2.2. Typical test architectures

The following architectures are examples where the use of test result analysis to the protocol specification seems particularly useful:

(a) A trace analysis module in a lower tester (see Figure 1) seems to be useful for certain non-standard test cases, as for instance during the debugging phase, or for testing the IUT for its behavior for cases which are not defined in the specification, e.g. "unexpected" input.

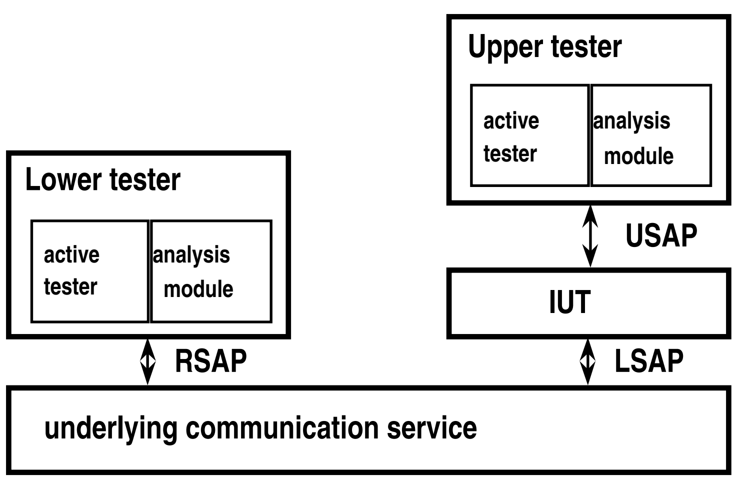


Fig. 1 - Distributed Test Architecture

(b) Figure 2 shows a test architecture for conformance resolution testing where two interworking protocol implementation are tested against one another. The arbiter shown in the Figure has to determine whether one of the IUT's shows any deviation from the protocol specification. For this purpose, the arbiter contains two trace analysis modules, each identical to the one for a lower tester under point (a) above, one corresponding to each IUT. In the case of tests over a local area network (LAN), an arbiter may be located directly over the LAN access protocol within one of the stations connected to the LAN (assuming that all packets passing through the LAN, also those destined for different addresses, can be received by that station) [Ayac 79, Molv 85]. In the case of communication over a long distance network (WAN), assumed in Figure 2, the observer may be located in a separate computer which is accessible through the network at a particular address.

(c) In the case of the "ferry" [Zeng 85] or "astride" [Rafi 85] test architectures, the service primitive interactions of the IUT are remotely controlled by the tester, as shown in Figure 3. In this case, the active tester controls all input to the IUT and the trace analysis module observes all interactions of the IUT, although the relative order of interactions at the lower and upper interfaces of the IUT, respectively, is not known to the tester because of the communication delays between the tester and the IUT. (This corresponds to a global trace observer which receives the local trace from each interface of the IUT; called "communication level (2)" in [Boch 89k]). This architecture can be used for the same purposes as architecture (a). The advantage is that the error detection power of the trace analysis module is much higher, since both interfaces of the IUT are observed. However, certain errors related to the relative timing between interactions at the upper and lower interfaces cannot be detected with certainty (see [Dsso 86b, Boch 89k] for more details).

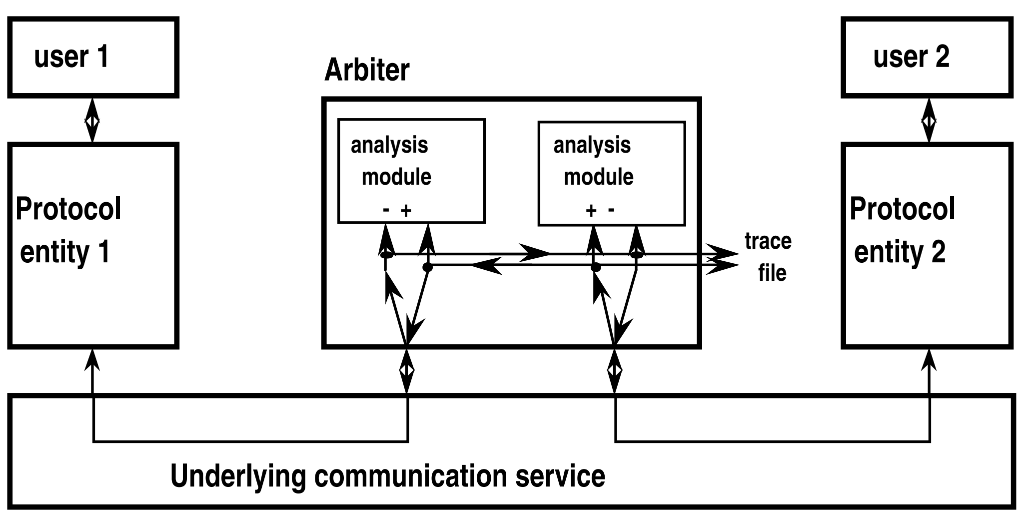


Fig. 2 - Test architecture with arbiter.

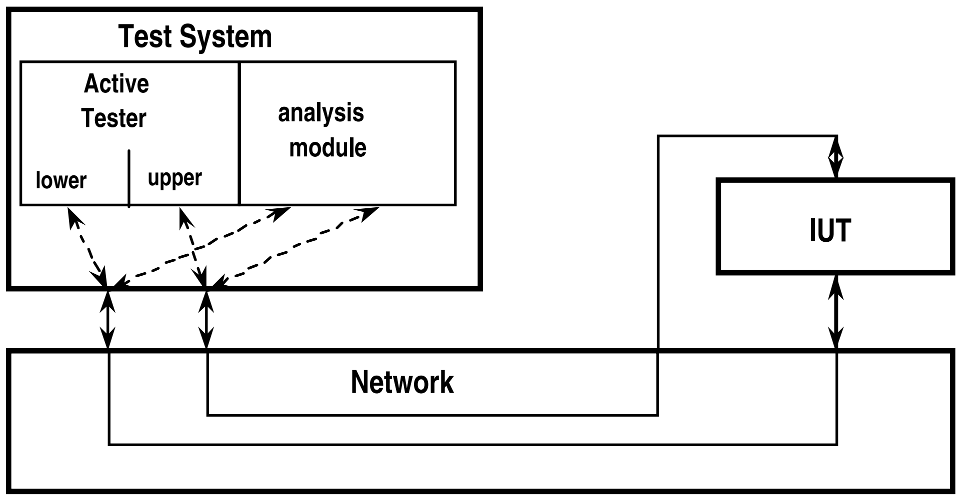


Fig. 3 - "Ferry" or "astride" test architecture.

3. Trace analysis with respect to LOTOS specifications

We describe in this section how an observed execution trace can be validated against a reference specification written in LOTOS. As explained in [Boch 89k], a given reference specification can be transformed into the specification of the corresponding trace analysis module by transforming input into output and taking into account the (possibly) non-deterministic nature of the reference specification. In this paper, we describe a slightly different approach where the reference specification is not modified, but directly interpreted by the test trace analyzer, together with the observed trace of interactions.

We assume that the observed trace of interactions has been translated into the form of a sequential LOTOS behavior expression. In the case of Transport protocol PDU's, as considered below, routines for translation between the coded PDU's and the corresponding LOTOS format shown below must be written. In the case of Application layer protocols using the ASN1 encoding scheme, the generation of translation routines can be automated if a standard translation scheme from ASN1 definitions to LOTOS data type definitions is adopted [Boch 89h]. Difficulties related to the translation of various description techniques for data structures are also discussed in [Mack 88].

For the example of a simplified Transport specification, as described in [Boch 87k], the following LOTOS interaction sequence represents a trace observed in the ferry architecture of Figure 3, where the lower tester initiates the establishment of a connection. It corresponds to the first part of the basic test case for Transport protocol conformance testing described in [Sari 86b].

```
Process  T [ TS, NS ] : noexit :=

(* a CR PDU arrives over the network service access point NS from the remote entity *)
NS      !NCEP_id_1
        !make_NDATAind(encode(make_PDU(CR_PDU
          (T_suffix_1,T_suffix_1,Insert(expedited_data,{}),
          NatNum(++(Dec(1),Dec(0)))),Succ(0),Succ(0))),true);

(* A connect indication "TCONind" is forwarded to the local user over the Transport
service access point TS *)
 TS     !T_suffix_1
        !TCEP_id_1
        !make_TCONind(make_T_addr(find_remote_N_addr,T_suffix_1),
          Insert(expedited_data,{}));

(* the user responds by a connect response "TCONresp" *)
 TS     !T_suffix_1
        !TCEP_id_1  !make_TCONresp(Insert(expedited_data,{}));

(* a CC PDU is returned to the remote Transport entity *)
 NS     !NCEP_id_1
        !make_NDATAreq(encode(make_PDU(CC_PDU(Insert(
          expedited_data,{}),Succ(Succ(Succ(Succ(Succ(Succ(Succ
          (Succ(Succ(Succ(0)))))))))),Succ(0),Succ(0))),true);

(* a data PDU is received from the remote entity and forwarded to the local user *)
 NS     !NCEP_id_1
        !make_NDATAind(encode(make_PDU(DT_PDU(0,Octet(Octet(
          1,1,1,1,1,1,1,1)),true),Succ(0),Succ(0))),true);


 TS     !T_suffix_1
        !TCEP_id_1
        !make_TDATAind(Octet(1,1,1,1,1,1,1,1)),true);

endproc
```

In general, a given trace T[g1, ..., gn] is valid to a reference specification S[g1, ..., gn], if the specification allows for an execution history which has a visible trace equal to T. One may also consider a LOTOS behavior expression of the form

T[g1, ..., gn] |[g1, ..., gn]| S[g1, ..., gn]

In the case that T is not valid, this expression will always lead to a deadlock before all interactions of T have been executed.

We have modified an existing LOTOS interpreter [Logr 88] in such a way that it checks whether a given trace of observable interactions is valid to a given reference specification. The existing interpreter requires interactions with the user for the selection of the next event to be interpreted, which may be an observable interaction at the gates g1 through gn, an event i, or an internal interaction at one of the gates hidden by the specification. Another version of the interpreter generates an overview of all possible sequences of observable interactions by backtracking over all possible internal events in case that several non-deterministic choices exist [Guim 89]. However, this latter version does not handle any interaction parameters. Our modified interpreter for test trace analysis, called TETRA, takes interaction parameters into account and uses backtracking to determine whether the tree of possible execution histories defined by the specification includes a history which gives rise to the trace T of observed interactions.

Backtracking is necessary for all those occasions where the reference specification allows for non-deterministic choices which are not directly visible. These choices relate to one of the following cases:

(a) Execution of internal events, i.e. event i or internal interactions on hidden gates, which appear as alternatives within a choice of subexpressions.

(b) Idem, when appearing within a CHOICE statement relating to alternative gates.

(c) Selection of a data value associated with a CHOICE statement.

(d) Expansion of recursive definitions in case of non well-guarded expressions, i.e. alternatives not beginning with an interaction or internal event.

In the cases (a) and (b), only a finite (usually small) number of alternatives are available, and they can be explored by the trace analysis interpreter, through backtracking, without excessive loss of efficiency. For case (c), however, the number of possible choices is sometimes not even bounded, as for instance in the case of the choice of a natural number. The trace analyzer has to determine whether a suitable choice of value exists which makes the given trace acceptable by the specification. A not completely satisfying approach is to arbitrarily limit the range of the choices considered by the interpreter. It is also necessary to define a procedure for enumerating the values of the corresponding data type.

The case of non well-guarded expressions (point (d) above) is more difficult to handle. It seems that the approach for interpreting non well-guarded expressions described in [Wu 89] is also applicable to trace analysis and could be used to handle the cases (c) and (d) above.

We present in the following a simple example to demonstrate some of these issues.

Example 1

```
(1)     specification   S1 [a,b,c,d,e,f,k] : noexit :=
(2)     Library   BasicNaturalNumber   endlib
(3)     behaviour
(4)        P[a,b,c,d,e,f,k]
(5)     where
(6)         process  P[a,b,c,d,e,f,k] : noexit :=
(7)             a?x:Nat [x=Succ(0)]; b;
```

```
                                   (     c; k; stop
                                    []   c; i; (    d; e; stop
                                                []  d; a; stop
                                              )
                                    )
                 []   a!Succ(0); a; (      b; k; stop
                                      []   b; c; e; stop
                            )
                 []   b!0; k; (       e; stop
                           []    d; c; stop
                           []    f!Succ(0); a; stop
                          )
          endproc
     (*------------------  Traces ------------------*)
      process  T1 [a,b,c,d,e,f,k] : noexit :=
            b!0; k; d; c; stop                    endproc
      process  T2 [a,b,c,d,e,f,k] : noexit :=
            a!Succ(0); b; c; d; e; stop           endproc
      process  T3 [a,b,c,d,e,f,k] : noexit :=
            b!0; k; f; d; stop                    endproc
     endspec
```

The above specification can be represented by the following tree (Figure 4):

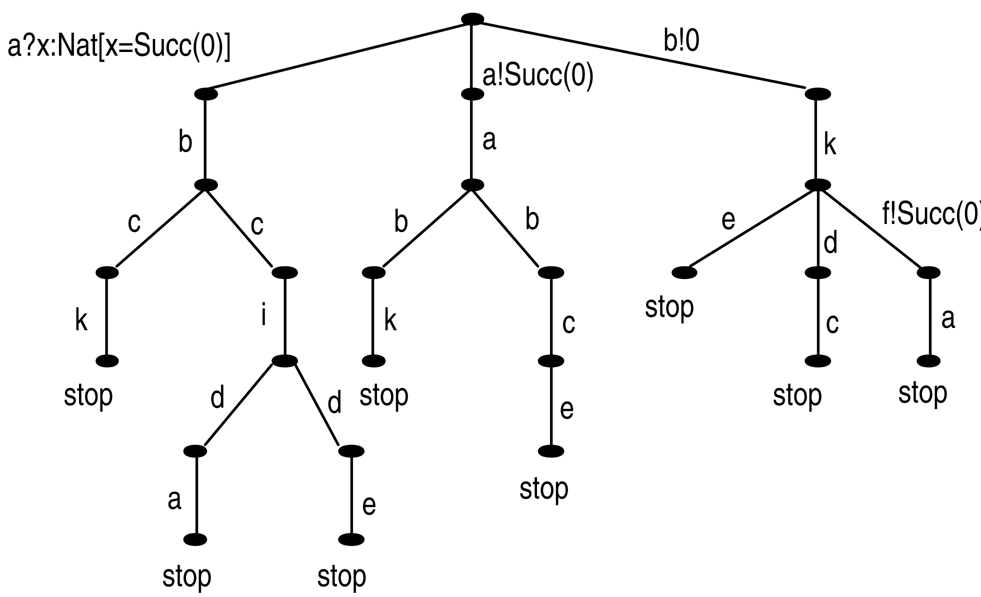
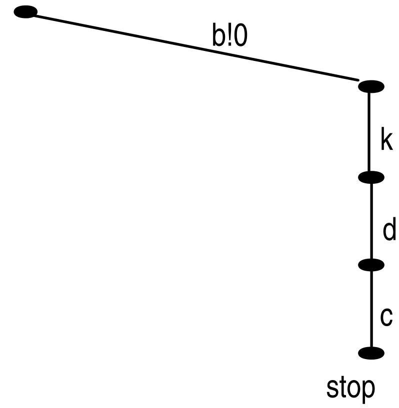
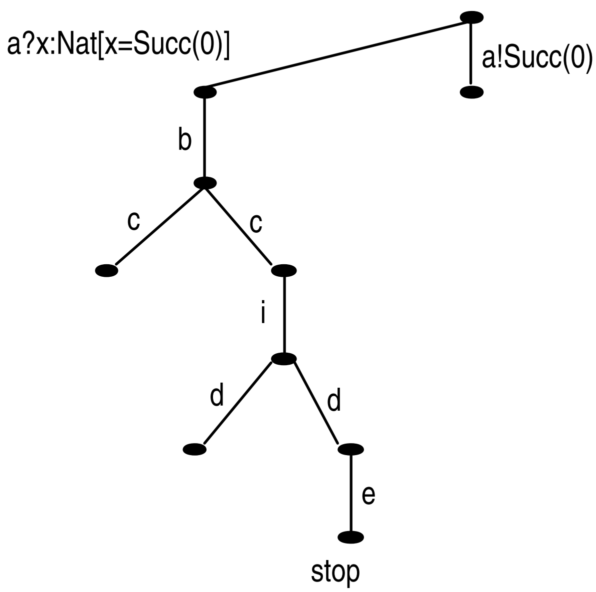


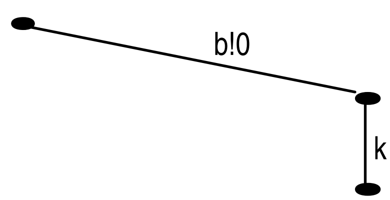
Figure 4: Tree representation of the specification of Example 1.



(a) Tree representation of T1 | | S1



(b) Tree representation of T2 | | S1



(c) Tree representation of T3 | | S1

Figure 5: Tree representation of trace analysis.

The parallel composition of the first trace T1 with the specification is represented by the subtree of Figure 5(a). Only one branch is possible, and there is no non-determinism.

Figure 5(b) represents the parallel composition of the second trace T2 with the specification. The trace is valid according to the specification, and there are some points with non-deterministic choice.

The parallel composition of the third trace T3 gives rise to the tree of Figure 5(c) with only two actions, although the trace includes four. The trace is not accepted by the specification.

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.

The TETRA test trace analyzer 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 which is consistent with the given trace and reference specification 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 for which the analyzer provides diagnostics 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.

The test trace analyzer 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 observed trace 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 behavior tree one needs more than N errors to explain the observed trace, the 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 explain the trace to the given branch of the specification. Several different branches may therefore lead to different diagnostics for the same observed trace.

A simple example is given by the specification

```
process  S2 [a,b,c,d,e,k] : noexit :=
   a!Succ(0); (    c; k; a; b; stop
                  [] b; k; e; d; stop )
endproc
```

and the trace

```
process T4 [a,b,c,d,e,k] : noexit :=
   a!Succ(0); c; k; e; d; stop
endproc
```

For this example, the analyzer will first explore the first alternative of the specification and encounter a discrepancy with the specification at the fourth 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. Therefore the analyzer will backtrack and explore the second alternative of the specification. It finds that the observed trace can be explained with one wrong action at the second place, which is formulated as the diagnostic message "action 2 should be: b". 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 4 should be: a ; action 5 should be: b".

In this example, is seems intuitively clear that the error is the second interaction. However, in more complicated situations, it is less clear what the "error" is. In any case, the analyzer provides a number of possible diagnostic messages, each indicating how the observed trace could be obtained from a correct trace by a small number of changes. The second diagnostic above also shows that sometimes changes are indicated only several interactions after the "error" occurs.

A second type of errors considered by the test trace analyzer are additional and missing actions. Here the hypothesis is, that the observed trace includes either an interaction that should not be present according to the specification, or that 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 examples demonstrate the diagnostic function of the trace analyzer.

Example 2

Using the specification S1 of Example 1, we consider six traces which have "wrong actions".

```
process  T5 [a,b,c,d,e,f,k] : noexit :=
   b!0; a; d;c; stop   /* action 2: erroneous gate */          endproc

process  T6 [a,b,c,d,e,f,k] : noexit :=
   a!0; a; b; k; stop  /* action 1: erroneous parameter */     endproc
```

```
process  T7 [a,b,c,d,e,f,k] : noexit :=
      b; k; e; stop  /* action 1: missing parameter */                endproc

process  T8 [a,b,c,d,e,f,k] : noexit :=
      a!Succ(Succ(0)); b; c; d; e; stop  /* action 1: predicate evaluated to false */
endproc

process  T9 [a,b,c,d,e,f,k] : noexit :=
      b!0; a; d; e; stop  /* actions 2 and 4: erroneous gates */   endproc

process  T10 [a,b,c,d,e,f,k] : noexit :=
      a!Succ(0); k; f; k; stop  /* actions 2 et 3: erroneous gates;
                                   2 possible diagnostics */          endproc
```

The first phase of the analysis indicates that all these traces are rejected by the reference specification. The second phase is thus initiated to determine the reasons of non-conformance. For N = 1 (number of wrong actions to be considered) the system generates the following diagnostics.

<u>Analysis of trace T5:</u>

Diagnostic 1 :	action 2 should be:	k	** erroneous action gate **
	corresponding to lines (16) and (17) of the specification.		

<u>Analysis of trace T6:</u>

Diagnostic 1 :	action 1 should be:	a !Succ(0)	** erroneous action parameter **
	line (13).		

<u>Analysis of trace T7:</u>

Diagnostic 1 :	action 1 should be:	b !0:Nat	** erroneous action parameter **
	line (16).		

<u>Analysis of trace T8:</u>

Diagnostic 1 :	action 1 should be:	a !Succ(0)	** predicate evaluated to false**
	lines (7) and (9).		

For the traces T9 and T10 the system gives no diagnostics since they cannot be explained by a single error. For N = 2, however, it generates the following diagnostics.

<u>Analysis of trace T9:</u>

Diagnostic 1 :	action 2 should be:	k	** erroneous action gate **
and	action 4 should be:	c	** erroneous action gate **
	lines (16) and (17).		

<u>Analysis of trace T10:</u>

Diagnostic 1 :	action 2 should be:	b	** erroneous action gate **
and	action 3 should be:	c	** erroneous action gate **
	lines (7) and (8).		
Diagnostic 2 :	action 2 should be:	a	** erroneous action gate **
and	action 3 should be:	b	** erroneous action gate **
	line (13).		

Example 3

Using again the specification S1 of Example 1, the following traces give rise to diagnostics involving missing or additional actions.

Analysis of trace: a!Succ(0); k; a; b; c; stop with N = 1:
Diagnostic 1 : action 2 should be absent lines (13) and (14).

Analysis of trace: b!0; b!0; k; e; stop with N = 1:
Diagnostic 1 : action 2 should be absent line (16).

Analysis of trace: a!Succ(0); b; c; k; f; stop with N =1:
Diagnostic 1 : action 5 should be absent lines (7) and (8).

Analysis of trace: b!0; k; f; e; d; stop with N = 1: No diagnostic
Idem, for N = 2:
Diagnostic 1 : actions 3 and 5 should be absent line (16).
Diagnostic 2 : actions 3 and 4 should be absent lines (16) and (17)

Analysis of trace a!Succ(0); b; k; stop for N = 1:
Diagnostic 1 : action ’c’ is missing, should be in position 3 lines (7) and (8).
Diagnostic 2 : action ’a’ is missing, should be in position 2 line (13).

Analysis of trace a!Succ(0); c; e; stop for N = 1: No diagnostic
Idem, for N = 2:
Diagnostic 1 : actions ’b’ and ’d’ are missing,
should be in position 2 and 4 resp. lines (7) and (9).
Diagnostic 2 : actions ’a’ and ’b’ are missing
should be in position 2 and 3 resp. lines (13) and (14).

Example 4

In this example we perform, for each trace, the three kinds of diagnostic analysis. This gives rise to a mixture of diagnostics.

```
(1)    specification  S3 [a,b,c,d,e,f,k]  : noexit
(2)    Library  BasicNaturalNumber  endlib
(3)    behaviour
(4)    P[a,b,c,d,e,f,k]
(5)    where
(5)       process P[a,b,c,d,e,f,k]  : noexit :=
(6)            a?x:Nat [x = Succ(0)]; b;
(7)              (   c; k; f; stop
(8)              [] c; i; d; e; stop
(9)              )
(10)           []  a!Succ(0); a; (   k; d; stop
(11)                              [] b; c; k; stop
(12)                             )
(13)           []  b!0; k; (   e; stop
```

```
(14)                    [] d; c; stop
(15)                    [] f!Succ(0); a; stop
(16)                   )
(18)      endproc
     (*------------------ Traces ---------------------*)
      process  T1 [a,b,c,d,e,f,k]:noexit:=
        a!Succ(0); a; c; k; stop       endproc

      process  T2 [a,b,c,d,e,f,k]:noexit:=
        b!0; k; d; e; stop             endproc
     endspec
```

<u>Analysis of trace T1:</u> for N = 1,
Diagnostic 1: action 2 should be: b ** erroneous action gate **, lines (6) and (7)
Diagnostic 2: action 3 should be absent, line (10)
Diagnostic 3: action ’b’ is missing, should be in position 3, lines (10) and (11).

<u>Analysis of trace T2:</u> for N = 1,
Diagnostic 1: action 4 should be: c ** erroneous action gate **, lines (13), (14).
Diagnostic 2: action 3 should be absent, line (13).
Diagnostic 3: action 4 should be absent, lines (13) and (14).

5. Conclusions

We believe that the automatic analysis of test results with respect to the protocol specification, as described in this paper, is an interesting alternative to test cases with predefined verdicts, since it can be used in situations where standardized test cases are not practical. The paper presents an operational test trace analysis system which compares the observed test results with a reference specification written in LOTOS. In the case of non-conformance with the specification, the system also provides diagnostics which may be useful for locating the fault(s) that caused the error in the observed trace.

Further work is required to evaluate the feasibility of the described approach in the case of complete specifications of OSI protocols, which are much larger than the here described examples. In addition, it is also not clear to what extent this approach is efficient enough to be used for on-line analysis of test results.

The test trace analysis system described here handles most aspects of LOTOS specifications. However, further research is required for handling certain types of specifications with non well-guarded expressions or general CHOICE statements.

It is important to note that a similar analysis approach can be used to validate the verdicts of standardized OSI test cases to the corresponding protocol specification. For this purpose, it is necessary, however, to have a reference specification written in an FDT and a translation of the test cases into a form suitable for the analysis. Some preliminary experiments of this kind have been performed [Bell 89] using an approach similar to the trace analysis described above.

Acknowledgements: We thank Fayez Saba for providing the trace example for the simplified Transport protocol and helping debugging an early version of TETRA. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada, the Ministry of Education of Quebec, and the Ministry of Higher Education of Algeria.

References

[Ayac 79] J.M.Ayache, P.Azema, M.Diaz, "Observer: a concept for on-line detection for control errors in concurrent systems", 9-th Int. Symp. FTC, Madison, June 1979.

[Bell 89] O.B.Bellal, "Analyse automatique de résultats de tests appliquée aux protocoles de communication", Master's thesis, Dept. d'IRO, Univ. de Montréal, 1989.

[Boch 87k] G.v.Bochmann, "Specifications of a simplified Transport protocol using different formal description techniques", submitted to Computer Networks and ISDN Systems.

[Boch 89h] G.v.Bochmann and M.Deslauriers, "Combining ASN1 support with the LOTOS language", Proc. IFIP Symp. on Protocol Specification, Testing and Verification XI, June 1989, North Holland Publ.

[Boch 89k] G.v.Bochmann, C.He, D.Ouimet, and R.Zhao, "Protocol testing using automatic trace analysis", to be presented at Canadian Conf. on Electrical and Computer Engineering, Sept. 1989.

[Dsso 85] R.Dssouli, G.v.Bochmann, "Error detection with multiple observers", Proc. IFIP Workshop on Protocol Specification, Testing and Verification, Toulouse, France, June 1985.

[Dsso 86b] R.Dssouli, "Etude des méthodes de test pour les implantations de protocoles de communication basées sur les spécifications formelles", PhD thesis, Université de Montréal, 1986.

[Guim 89] R.Guillemot, L.Logrippo, "Derivation of useful execution trees from LOTOS specifications by using an interpreter", in: K.J.Turner (ed.) Formal Description Techniques, North-Holland, 1989. (Proc. of the first FORTE International Conference, Stirling, UK, 1988, pp. 311-325).

[Logr 88] L.Logrippo, et al., "An interpreter for LOTOS: A specification language for distributed systems", Software Practice and Experience, to appear.

[Loto 87] ISO DIS8807 (1987), "LOTOS: a formal description technique".

[Mack 88] L. Mackert et al., "A generalized conformance test tool for communication protocols", Proc. Int. Conf. on Distributed Computing Systems, IEEE, San José, 1988.

[Molv 85] R.Molva, M.Diaz, J.M.Ayache, "Observer: a run-time checking tool for local area networks", 5-th IFIP Workshop on Protocol Specification, Testing and Verification, Toulouse, 1985, North-Holland.

[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 C] ISO TC97/SC21, DP 9646/1 and DP 9646/2: OSI Conformance methodology and framework, Part 1: General Concepts, Part 2: Abstract Test Suite Specification, 1987.

[Rafi 85] D. Rafiq, R. Castanet, C.Chraibi, j.P.Goursaud, J.Haddad, X.Perdu, "Towards and environment for testing OSI protocols", IFIP Workshop on Protocol Specification, Verification and Testing (Toulouse, 1985), North Holland.

[Rafi 85b] O.Rafiq, "Tools and methodology for testing OSI protocol entities", Proc. Int. Symp. on Fault Tolerant Comp., FTCS 15, IEEE, Ann Arbor, 1985.

[Sari 86b] B.Sarikaya, G.v.Bochmann, M.Maksud, J.M.Serre, "Formal specification based conformance testing", Proc. ACM SIGCOMM Symposium, Aug. 1986, pp. 236-240.

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

[Wu 89] C.Wu and G.v.Bochmann, "An execution model for LOTOS specifications", submitted for publication.

[Zeng 85] H.X.Zeng, D.Rayner, "The impact of the ferry concept on protocol testing", in Protocol Specification, Testing and Verification (IFIP Workshop), M.Diaz (ed.), North Holland, 1986, pp. 533-544.