13

A rigorous and practical approach to service testing

L. Feijs.
Philips Research Laboratories,
Eindhoven University of Technology.
Address: Philips Research, Prof. Holstlaan 4, 5656AA Eindhoven,
Tel. +31 402742953, Email: feijs@natlab.research.philips.com

M. Jumelet.
University of Amsterdam.
Present affiliation: Data Sciences B.V. Amsterdam.

Abstract

We present an approach for the systematic development of abstract test cases for service testing. The approach is rigorous in the sense that the abstract test cases are derived from a rigorous specification. The approach is practical in the sense that it contains guidelines for choosing amongst the many possible execution paths. The method has been developed in practice. Tools are used for interactive simulation and for certain translation steps.

Keywords

Protocols, testing, TTCN, PSF, MSC, SDL.

1 INTRODUCTION

Distributed systems are most often organised along two more or less orthogonal decomposition principles: distribution and layering. Testing the components of distributed systems is an important issue, but because there are two decomposition principles, there are several kinds of testing (such as conformance testing and service testing). Much work has been done on protocol conformance testing, which is about testing protocol entities, that is, the implementations which are at the intersection of these two principles (a protocol entity is rougly speaking the local implementation dealing with one layer). Conformance testing is of key relevance for the correct operation of open systems, in particular if there are many different implementors.

On the other hand, there are many situations in which there are no multiple vendors (or not yet). Moreover the service to be provided by some layer may be completely new; in this case it is likely that priority is given to get this service as soon as possible properly implemented and tested and that the conformance issues are left for a later phase.

Therefore we believe it is appropriate to address the issue of 'service testing'. Service testing presents certain technical difficulties which make it different from conformance testing. In particular the state space of the system is the product of the state spaces of the components. The specification of a service is more difficult than the specification of a single protocol entity. Certain concepts and techniques originally developed for conformance testing can be fruitfully applied to service testing too (with adaptations).

Moreover it is well known (Knightson 1993) that in practice conformance does not imply interoperability. Amongst several reasons, this is because in practice, testing can not be exhaustive. The term 'interoperability testing' is often used for the trials to see if the system as a whole really works. This is considered supplementary to conformance testing.

We use the term 'service testing' to stress our point of view that it is more than a supplement to fill the holes left by conformance testing, and that service testing has a value in its own right. Service testing deals with the compatibility between the artifacts created by the decomposition principle of layering (the layers), without assumptions about the protocols that implement a layer. The service is tested against a service specification.

We use terminology from the world of conformance testing (Knightson 1993). Some additional terms will be proposed for the purposes of service testing.

2 SURVEY OF THE TEST DEVELOPMENT METHOD

The approach to develop abstract test cases is rigorous (the abstract test cases are derived from a rigorous specification) and practical (it contains guidelines for choosing execution paths). The main steps of the method are divided into two groups: (1) preparation steps, relevant for the whole test suite, and (2) steps to be done for each test goal. The following steps must be done once:

- rigorous specification of the service, using a formalism which allows simulation. This serves as a model for the external behaviour of the service (not of the real implementation). We used the process specification formalism PSF (Mauw, Veltink 1993).
- making the test architecture explicit.
- extraction of a finite state diagram, which we call the service state transition diagram (S-STD). It serves as a model of the service as perceived by one local user.
- performing a further reduction of the S-STD, keeping only particular states, which we call 'stable states' and distinguishing the user rôles of Initiator and Follower. The resulting diagram is called rôle state transition diagram (R-STD).
- establishing test goals, in particular by choosing the ability to sucessfully perform certain transitions (one goal for each transition from the Follower's R-STD).

The following steps must be done for each test goal:

- draw a rôle transition path (RTP), which is an outline of two or more simultaneous walks through the R-STDs. This is needed as a guidance when interactively performing the next step (simulation).
- simulate a 'straight' execution, by which we mean a sequence of events in which the service behaves as expected.

- construct a sequence chart (SC) from the simulation trace. The SC serves for documenting the test (useful when analysing log files later and for debugging). Moreover the SC is used as an intermediate format before making the next step.
- construct an initial TTCN description. This is essentially the same as the 'straight' execution, with alternatives '`?OTHERWISE FAIL`' added for each step.
- simulate alternatives: in general the service can have internal non-determinism, most of which is because the service model also describes the behaviour under 'bad conditions' (i.e. the underlying medium or network fails).
- complete the TTCN description by assigning verdicts `PASS` or `INCONC` to the alternatives and editing the initial TTCN description.

The SC formalism used is a simplified (synchronous) version of MSC (CCITT 1992); within Philips such SC used to be indicated by the somewhat confusing term 'interworkings'. Test execution is outside the scope of this paper; for a few remarks on it see Section 15.

3 EXAMPLE OF A SERVICE

In this section we will sketch a service offered by a small hypothetical system. It is a true toy example, devised in order to explain our test approach; it is kept deliberately simple in order to fit in the scope of the present report. While working on this paper we discovered that this kind of service is more realistic than we initially thought, because the Wehkamp vending service (Wehkamp 1996) has some similar characteristics.

The service is a distributed service, as shown in Figure 1. There is a service interface which runs as a horizontal line through the architecture. If it is our goal to test this service, the implementation under test (IUT) is the shaded area in Figure 1.

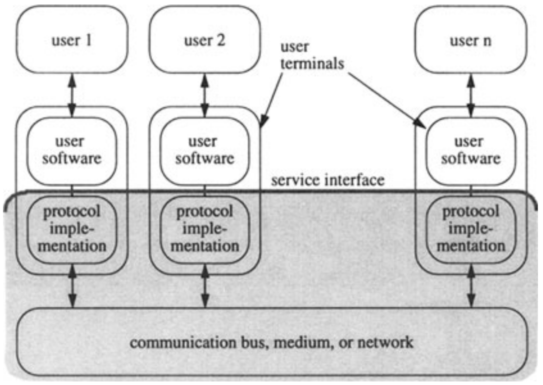

Figure 1: Typical System Architecture.

In our toy example, the service is meant to support processes of negotiating between two or more users about the price of certain objects. One user can offer some object (for example a painting or some other piece of art) and associate a price with it. Later, as time passes, he may decide to decrease the current price. Other users can bid for the object and later increase there current bid. If the system finds a match, the object is considered sold and the two users involved are informed. The abstract service primitives (ASPs) are:

1. offer(object,price): offer an object and require that at least 'price' (in some agreed unit currency like thousands of Guilders) be paid for it.
2. decrease(N): decrease the required price by N (only for a user who offered an object).
3. flush(): remove all bidders in order to start the negotiation process over again (only for a user who has offered an object at present)
4. bid(object,price): try to acquire 'object' for 'price'. The mechanisms to find out whether indeed someone is offering 'object' are not included in the service.
5. increase(N): increase the required price by N (only for a user who is bidding).
6. stop(): go to the idle state (withdraw offer or bidding).
7. ok(): indication from the service to the user to confirm 'offer' and 'bid'.
8. notok(): indication from the service to inform the user that a previous command offer, or 'bid' is not accepted. Also used to inform a user that a previous command like 'decrease', 'flush', or 'increase' was issued in a state not suitable for execution of that command.
9. sold(price): indication from the service to inform both an offering and a bidding user that the current object has been sold for 'price' (this example does not allow for simultaneous offering of objects, which of course could be modeled).

When specifying a service like this, it is often fruitful to consider it as a single process (even if it is distributed in reality). This view is used in Figure 2, where we show one typical run of the system. There are two users, called Jelle and Sjeng (two typical Dutch names) who are negotiating for a Degas (a typical impressionist French painting). In the run shown, the answer 'sold(4)' is sent to Jelle first, but the other order, where Jeng gets his 'sold(4)' answer first is possible too (this does not follow from Figure 2, but such facts could follow from a rigorous specification as in Section 4).

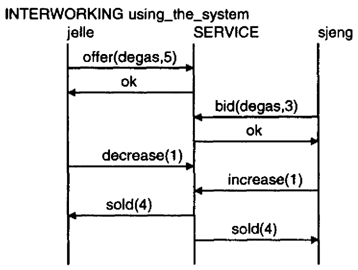

Figure 2: Example run of the system.

4 STEP: RIGOROUS SPECIFICATION OF THE SERVICE

This section is about the rigorous service specification, using PSF (Mauw, Veltink 1993). Type checking and simulation tools for PSF are publicly available. It is based on algebraic principles: signatures and equations, both for data types and for processes. It builds upon the theory of initial algebras for data types and upon the algebra of communicating processes (ACP) for the processes.

The advantage of having a rigorous specification is that it is much more precise than an informal description and that the simulator can be used to generate event-traces which are automatically correct. There is also a price to be paid for this: writing and reading rigorous specifications, requires certain skills and significant investments. We use a number of additional diagrams to make the model as readable and intuitive as possible: (S-STD, R-STD, SC, cf. Figures 5, 6 and 2 respectively)

The service interface can be viewed as a kind of application programmer's interface (API) which hides a communication mechanism and which can be used as a basis for building a system with user interface for object-offering and bidding. The atomic actions `c` and `a` model the commands and the answers which are sent to the service and received from the service, respectively. The commands and the answers are parameterised: their first parameter tells which user performs the request or gets the answer. The second parameter carries the real information of the abstract service primitive. The entire service interface layer is modeled as a single process `S` which accepts service requests (commands) from the users and gives answers to the users. It has an additional parameter of a type called `USERLIST`. This parameter is carried along for every recursive invocation of S, and in a certain sense, it contains the state of the service layer as a whole.

The PSF specification imports the data types used (`Nat`, `Obj`, `User`, `Bool`, `Info`, `State`, `UserList`). The internal state of the service is modeled as a list of four-tuples, one for each user (called Peter, Karel, Sjeng and Jelle). For each user the state contains, apart from the user-identity, (1) its main state (`idle`, `opening`, `offering` etc.), (2) its current price proposal (a natural number), (3) its object of interest (either for selling or buying). The specification is fixed for a number of four users. It is obvious how to extend this to any arbitrary number of users. The initial process is specified as:

```
S( cons(peter,idle,0,NONE
 , cons(karel,idle,0,NONE
 , cons(sjeng,idle,0,NONE
 , cons(jelle,idle,0,NONE,empty))))
 )
```

The rest of the specification is one big equation for `S(L)`. The right-hand side of this equation extends over several pages (of which we show a small fragment), since it is a sum of many terms: four terms about the events which can happen in state `idle`, three terms about the events which can happen in state `opening`, and so on.

We show the term which describes what happens if user U offers object (painting) P for a price of N, provided this user is in state idle. The equation must be valid for all alternative values of U,P and N and therefore the term is described using three `sums`. The `sum` construct generalises the `+` operator of alternative composition (choice). Please read `sum( U in USERs, xyz(U))` as $\sum_{U \in \mathrm{USERS}} \mathtt{xyz}(U)$ which is the same as `xyz(peter) + xyz(karel) + xyz(sjeng) + xyz(jelle)`.

There are a number of auxiliary functions which work on data types (specified algebraically but not included here). For a user U and state-list L, `state`(U,L) can be used to look up the individual state of this user. There are also three 'assignment' functions: `setobject, setstate, and setcurrent`. For example `setobject`(U,P,L) changes the object of interest for user U into P in the state-list L, returning a modified state-list.

After commanding an `offer`(P,N), a user is transferred to state `opening` (where it will be sorted out if he will get an `ok` or a `notok`). After commanding a `bid`(P,N), a user is transferred to state `trying`. After commanding a `stop` in state `idle`, nothing happens.

```
S(L) = sum( U in USERs,
       sum( P in OBJs,
       sum( N in NATs,
            [state(U,L)=idle] ->
            [eq(P,NONE)=false]->
            c(U,offer(P,N)) . S( setobject(U,P
                               , setstate(U,opening
                               , setcurrent(U,N,L)))
         )))                   ) +

       sum( U in USERs,
       sum( P in OBJs,
       sum( N in NATs,
            [state(U,L)=idle] ->
            [eq(P,NONE)=false]->
            c(U,bid(P,N)) . S( setobject(U,P
                             , setstate(U,trying
                             , setcurrent(U,N,L)))
         )))                 ) +

       sum( U in USERs,
            [state(U,L)=idle] ->
            c(U,stop) . S(L)
         ) +

       sum( U in USERs, sum( P in OBJs, sum( N in NATs,
            [state(U,L)=idle] ->
            ( c(U,decrease(N)) . a(U,notok) . S(L)
            + c(U,flush)       . a(U,notok) . S(L)
            + c(U,increase(N)) . a(U,notok) . S(L)
         ) )            )              ) +
```

The terms for opening, `offering`, `trying`, `bidding` and `finalising` are not included.

5 STEP: MAKING THE TEST ARCHITECTURE EXPLICIT

The test architecture can be different from the system architecture (like in Figure 1) which describes the system in normal use. A typical test architecture is shown in Figure 3. There is one special point of control and observation (PCO) which corresponds to the possibility of provoking 'bad conditions' (making the underlying communication bus, medium or network fail). Sometimes this must be done by a kind of manual intervention, e.g. by unplugging certain wires. In the present paper we assume the comfortable situation of a centralised test system. When testing the system, it is often fruitful to organise the tester

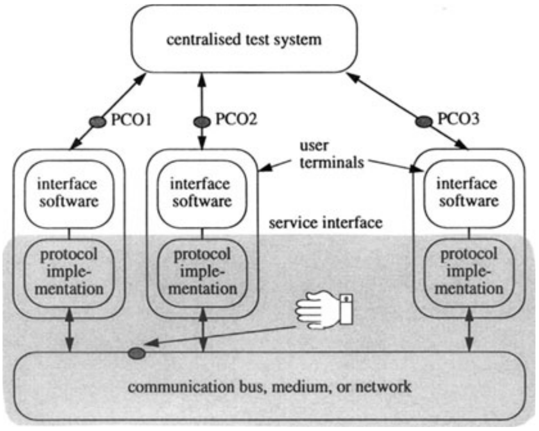

Figure 3: Typical Test Architecture.

as a single process. Because there are distinct PCOs for the devices, it is convenient now to consider each device as a single process. This view is used for example in Figure 4 below. The service is split into two processes, one for each user's PCO (so now 'jelle' and 'sjeng' do not refer to the real users, but to their respective terminals).

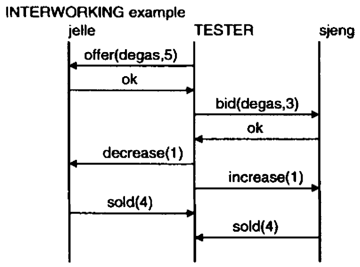

Figure 4: Test Run of the System.

6 STEP: EXTRACTION OF THE S-STD

The diagram of Figure 5 gives a survey of the entire service in a single diagram. We call this a service-STD (S-STD). The transitions are labeled with information about conditions, communications steps and internal actions. If possible this information is organised in two parts, separated by a horizontal line: the text above the line describes a command or a condition and the text below the line is the result (an answer or an internal action). In many projects this kind of diagram has been drawn already long before the test-phase of the system, since it is useful for other purposes as well. The diagram is compact and attractive from an intuitive point of view, and thus it is nice to have such a diagram next to the rigorous model.

The diagram is not rigorous: the descriptions of the conditions and the internal actions are informal and they leave room for all kinds of wrong interpretations.

In order not to complicate the diagram too much, certain actions which have no effect and which immediately return notok are not shown fully; by way of reminder there is just an unlabeled transition for the states in which such transitions are possible.

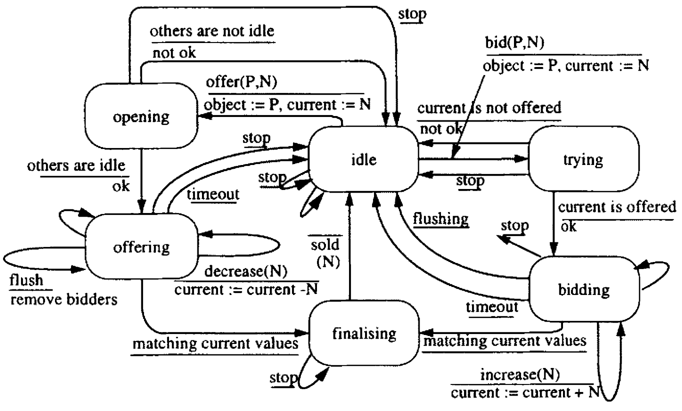

Figure 5: Finite State Diagram of the Service.

7 STEP: FROM S-STD TO R-STD

Next we perform a further reduction of the S-STD, keeping only particular states, which we call 'stable states' and distinguish the user rôles of Initiator and Follower. The resulting diagram is called rôle state transition diagram (R-STD). This is done twice: for the initiator and for the follower (Figure 6). We say that a state s is *stable* if whenever the system under test (SUT) is in this state, it will not move to another state s', unless by an external event. For example state trying is not stable because the service itself will find out that either an ok or a notok is to be issued. This definition of 'stable' is not completely rigorous; for example a stop command may force the transition from trying to `idle` too. But as a heuristic, the concept seems useful (tests will still be derived from simulation runs of the full PSF model and the loss of information in these diagram is therefore not harmful).

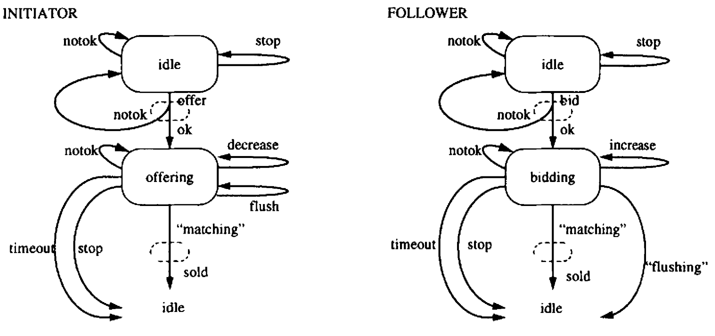

Figure 6: Role-State Transition Diagrams.

Note that `idle` is shown twice in each diagram; of course one occurrence is enough, but the present layout is convenient later, since test cases are drawn as walks trough the diagram. The present layout leads to walks which proceeds, roughly speaking, from top to bottom in the diagram.

8 STEP: ESTABLISH TEST GOALS

The R-STDs provide an excellent starting point for establishing the test goals. Of course one needs to have some idea about the degree of coverage required. Full test coverage is out of the question anyhow and the choice usually is a trade-off between the costs of testing on one hand and the costs of undetected defects on the other hand. Here we propose one particular choice which in some situations gives a reasonable compromise and which moreover provides some handles for managing the test development process. But of course other choices could be made here as well.

The criterion proposed here to make one abstract test case (ATC) for each transition of the Follower's R-STD. The test goals are numbered in Figure 7 (starting with the most important transitions). We add some explanation for the ATCs numbered 9, 10 and 11. These correspond to the commands `decrease`, `flush` and `increase`, all of which must yield `notok` as an answer. Similarly for 12, 13, 14 and 15 (`offer`, `decrease`, `flush` and `bid`). This has a number of advantages.

- the approach gives rise to a finite number of tests, 15 in this example (this number will increase according to the complexity of the service);
- all transitions will be executed at least once, so for certain classes of defects a complete coverage is achieved for the follower (if the service is not too complex of course); in particular these include the defect class where the transition is completely forgotten and bad code (opcode traps etc.) which causes a crash in an implementation;

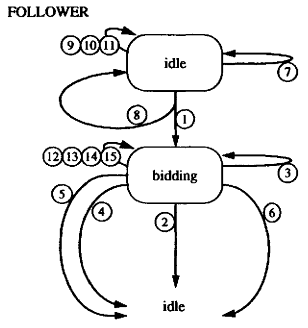

Figure 7: Survey of Test Goals.

- many services can be viewed upon as if their main purpose is that the initiator can bring the follower from one state to another; so, it is likely that a test campaign which leads the follower through all its transitions has the effect that at the end effectively the initiator has gone trough most of its transitions too (which can be checked, by marking each transition in the Initiators R-STP which is used in any of the ATCs of these 15 goals).

The followings steps are to be performed for each of the test goals.

9 STEP: DRAW A RÔLE TRANSITION PATH (RTP)

Let us demonstrate the development of ATC_2, which is the transition with condition "matching" and answer `sold`. In Figure 8 the RTP for the follower is shown (right-hand side). This path must be devised first. Based on this, one needs a certain path for the initiator in order to make the planned walk through the follower's R-STD happen. In practice it turned out convenient to draw these diagrams by hand, using a sufficient supply of paper copies of R-STD pairs (typically one initiator-follower pair on each sheet). For the coloring scheme we refer already to Figure 9. In order to have some heuristics for designing the follower's RTP we borrowed some concepts from the world of automated test generation for purposes of conformance testing, see e.g. (Sidhu, Leung 1989). We adapted them slightly; moreover we sometimes did not apply them completely rigorously according to the original definitions. As a heuristic and also for documentation purposes we found them very useful. Next to the concept of 'test-goal', which in our case coincides with a transition in the R-STD, we use three more concepts:

- transferring sequence, or preamble (a sequence of transitions in the R-STD leading from `idle` to the begin-state of the test goal).
- test goal (one of the transitions chosen in Section 8).
- unique input/output sequence (a sequence of commands and answers which uniquely distinguishes the end-state of the test goal from all other states - if possible).

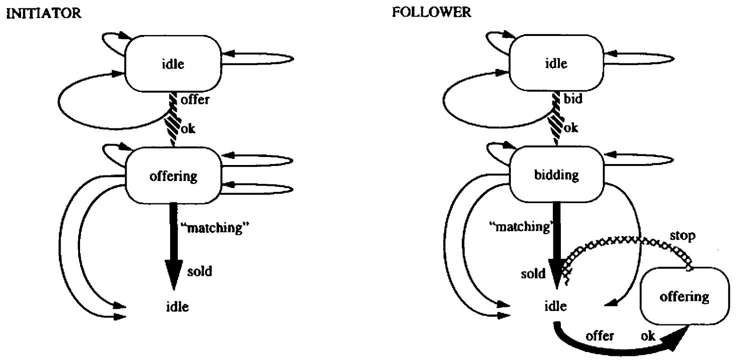

Figure 8: RTP for the Follower (and Initiator too) for ATC_2.

- synchronising sequence, or postamble (to return the system again in the `idle` state).

We used a kind of coloring scheme for indicating the four sub-segments of the RTPs. This is summarised in Figure 9. In practice it turned out convenient to use colored pens for this (green for the transferring sequence, red for the test goal, yellow for the unique input/output sequence, and blue for the synchronising sequence. For the RTPs of Figure 8

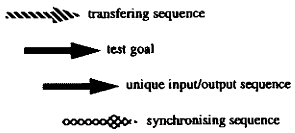

Figure 9: Coloring Scheme used in RTP.

the motivation is as follows. Since the goal is to see if the move from `bidding` to `idle` based on "matching" and with answer `sold` can be made properly, the follower needs to go to `bidding` first. Therefore the transferring sequence consists of one transition (`idle` $\xrightarrow{\text{bid, ok}}$ `bidding`) in the follower's R-STD (corresponding to two transitions in the S-STD). The test goal is one transition (`bidding` $\xrightarrow{\text{matching, sold}}$ `idle`) in the follower's R-STD (again corresponding to two transitions in the S-STD).

It is not sufficient that the proper answer (`sold`) has been obtained. It has to be checked whether the new state is `idle` indeed. This is done using a sequence of commands and answers which is able to uniquely identify `idle` (or at least excludes most other states). In this case it is not hard: `idle` can be characterised as the only state in which the command `offer` must be performed successfully (provided certain conditions concerning the other users hold). So the unique input/output sequence is (`idle` $\xrightarrow{\text{offer, ok}}$ `offering`). And in order to leave the system in an orderly state, ready for a next test, it is brought back to state `idle` by means of `stop`. So the synchronising sequence is (`offering` $\xrightarrow{\text{stop}}$

idle). Note that for the purpose of the unique input/output sequence, the follower has to switch to an initiator's role temporarily.

10 STEP: SIMULATE A 'STRAIGHT' EXECUTION

Next we use the PSF simulator to get a trace of the intended behaviour of the service. This can be done interactively; for each step, the simulator gives a clickable list of possible alternatives in a special menu window. The number of choices for each step is quite large: typically about 60 alternatives for this example service. The trace appears in another window, from which it is easily copied to a file for further processing.

This is demonstrated for ATC_2 again, using the RTP of Figure 8 as a road-map. Letting jelle play the initiator's rôle and sjeng the follower's rôle, and choosing to negotiate for a manet, the following trace was obtained:

```
atom c(jelle, offer(manet, s(0)))
atom a(jelle, ok)
atom c(sjeng, bid(manet, s(s(s(0)))))
atom a(sjeng, ok)
atom INT(sjeng, jelle)
atom a(jelle, sold(s(s(0))))
atom a(sjeng, sold(s(s(0))))
atom c(sjeng, offer(manet, s(s(s(s(s(0)))))))
atom a(sjeng, ok)
atom c(sjeng, stop)
```

This trace is correct by definition (if the PSF model is right); using the simulator one may make mistakes in the sense of not reading the road-map properly, but the trace found is still a valid trace for the given service (even it is not of help for the test goal).

11 STEP: FROM SIMULATION TRACE TO SC

By means of simple conversion tools, the trace can be transferred to a sequence chart in textual format. This is edited in order to make the begin-state and the end-state of the transition of the test goal explicit. Then there is another tool which makes a postcript file. The latter file only serves for documentation purposes. It is a useful help when trying to understand the TTCN-GR resulting in subsequent phases. As before, the service is split into two processes, one for each user's PCO, so 'jelle' and 'sjeng' do not refer to the real users, but to their respective terminals.

12 STEP: CONSTRUCT AN INITIAL TTCN DESCRIPTION

Since TTCN is the de facto standard for ATC description, we aim at producing graphical TTCN (TTCN-GR) next. It is not hard to construct an initial TTCN-GR description from the sequence chart automatically. We built a simple 'translator' from SC to TTCN-GR, programmed in Gofer. In general, the result is not a correct test yet, but it serves

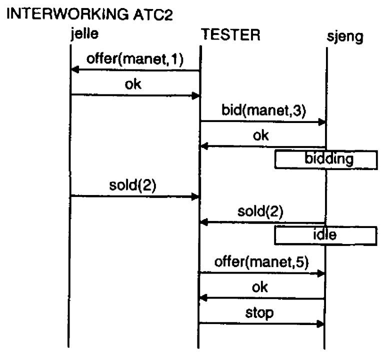

Figure 10: Sequence Chart of ATC_2.

as a draft for further development. E.g. we could remove some of the preliminary (PASS) verdicts and turn the last of them into PASS (and alternatives must be taken into account).

```
+----------------------------------------------------------------------------+
|Test Case ATC_2                                                             |
+----------------------------------------------------------------------------+
|Test Case Name : ATC_2                                                      |
|Group          : 1/                                                         |
|Purpose        : Transition from bidding to idle for follower (draft)       |
|Default        :                                                            |
|Comments       :                                                            |
+----------------------------------------------------------------------------+
|Nr | Label | Behaviour Description   | Constraints Ref | Verdict | Comments|
+----------------------------------------------------------------------------+
|0             jelle!offer(manet,1)                                          |
|1              jelle?ok                                  (PASS)             |
|2               sjeng!bid(manet,3)                                          |
|3                sjeng?ok                                (PASS)             |
|4                 jelle?sold(2)                          (PASS)             |
|5                  sjeng?sold(2)                         (PASS)             |
|6                   sjeng!offer(manet,5)                                    |
|7                    sjeng?ok                            (PASS)             |
|8                     sjeng!stop                                            |
|9                    sjeng?OTHERWISE                     (FAIL)             |
|10                 sjeng?OTHERWISE                       (FAIL)             |
|11                jelle?OTHERWISE                        (FAIL)             |
|12               sjeng?OTHERWISE                         (FAIL)             |
|13             jelle?OTHERWISE                           (FAIL)             |
+----------------------------------------------------------------------------+
```

13 STEP: SIMULATE ALTERNATIVES

The service may have internal non-determinism, or depend on timing properties or network states which cannot be fully controlled. So there are alternative behaviours, not captured yet. The simulator can be used to find such alternatives (they appear in the simulator's menu). Some intuition and extra information about the physical test circumstances is needed as input. So we run the simulator again, looking for alternatives. After line 3 (in the above TTCN-GR), `sjeng` could make the internal step 'timeout':

```
atom c(jelle, offer(manet, s(0)))
atom a(jelle, ok)
atom c(sjeng, bid(manet, s(s(0)))))
atom a(sjeng, ok)
atom int(sjeng)
```

This is acceptable behaviour, but it does not help in judging the outcome of the test goal, so it is judged `INCONC`. There is another branch: the service may give `sold` to `jelle` first and to `sjeng` next, or it may do so in the reverse order:

```
atom c(jelle, offer(manet, s(0)))
atom a(jelle, ok)
atom c(sjeng, bid(manet, s(s(0)))))
atom a(sjeng, ok)
atom INT(sjeng, jelle)
atom a(sjeng, sold(s(s(0))))
atom a(jelle, sold(s(s(0))))
atom c(sjeng, offer(manet, s(s(s(s(0)))))))
atom a(sjeng, ok)
atom c(sjeng, stop)
```

The generated TTCN will already accept this trace too, since the TTCN semantics assumes that the `sold` answer for `sjeng` is buffered until `jelle`'s `sold` answer is consumed.

14 STEP: COMPLETE THE TTCN DESCRIPTION

Several lines have to be added because of the alternatives discovered. A timer is added to find out if the `sold` answers do not happen. The timeouts are `INCONC`. Some of the preliminary verdicts generated first are removed; others are turned into final verdicts.

```
+-----------------------------------------------------------------------+
|Test Case ATC_2                                                        |
+-----------------------------------------------------------------------+
|Test Case Name : ATC_2                                                 |
|Group          : 1/                                                    |
|Purpose        : Transition from bidding to idle for follower          |
|Default        :                                                       |
|Comments       :                                                       |
+-----------------------------------------------------------------------+
|Nr | Label | Behaviour Description   | Constraints Ref | Verdict | Comments|
+---------------------------------------------------------------------------+
|0            jelle!offer(manet,1)                                          |
|1             jelle?ok                                                     |
|2              START T1 (10 sec)                                           |
|3               sjeng!bid(manet,3)                                         |
|4                sjeng?ok                                                  |
|5                 jelle?sold(2)                                            |
|6                  sjeng?sold(2)                                           |
|7                   sjeng!offer(manet,5)                                   |
|8                    sjeng?ok                            PASS              |
|9                     sjeng!stop                                           |
|10                   sjeng?OTHERWISE                     FAIL              |
|11                 ?TIMEOUT T1                           INCONC            |
|12                 sjeng?OTHERWISE                       FAIL              |
|13                ?TIMEOUT T1                            INCONC            |
|14                jelle?OTHERWISE                        FAIL              |
|15               sjeng?OTHERWISE                         FAIL              |
|16            jelle?OTHERWISE                            FAIL              |
+---------------------------------------------------------------------------+
```

15 TOOL ASPECTS AND TEST EXECUTION

The PSF simulator plays a central role in our approach. Probably one could use LOTOS or SDL simulators as well. The Philips 'Interworking Tool' (Mauw, Winter 1993) was used to make the diagrams of the type shown in 2, 4 and 10. STDs, R-STDs were made using standard drawing tools (Framemaker) and the RTPs were drawn by hand with color pencils (Framemaker for the paper). We built some simple preprocessing tools with cpp (macro preprocessor) and `lex` (scanner generator) to convert traces to sequence charts and a simple 'translator' from sequence charts to TTCN, programmed in Gofer. In our project the ATCs were manually translated to C and in this step several practical issues were resolved too, like accessing the PCOs and mapping the packet structure to the TTCN events. There are options for further automation; e.g. translating TTCN to C can be automated, maybe using ASN-1 for the message-layout.

16 OPEN ISSUES

In general, there may be more users and there may be more sub-services. In that case various extensions of the approach are possible. Due to space limitations we will not discuss them here.

There are a number of subtle technical problems which arise in practice because the synchronous communication model of PSF does not formally match the asynchronous model of TTCN, which need not perfectly match the typical event-based mechanisms used in a real service. We used intuitively right ad-hoc solutions for these issues and for the real-time aspects, but there are clearly options for further research and improvement.

17 CONCLUDING REMARKS

The proposed method has been developed in two phases when we had to generate tests for new services developed for a Philips product. In an earlier phase of this project, PSF had been used already (most of the PSF was not developed by the authors of the present paper).

The test-development activities proceeded in two phases. In the first phase we devised tests for one layer of service, which we did almost entirely by hand, checking only a few tests with the simulator (we had a PSF model for that service too). The tests were translated to C and applied to the implemented service. In the second phase (in the context of the same project), we addressed another more complex layer of service and we decided to better exploit the available specifications and tools. For this purpose we developed the Gofer translator from sequence charts to TTCN, and used simulation traces as the starting point for all tests. In this way we developed about 30 ATCs. Most of these tests were never put into execution because of shifting priorities and other boundary conditions of the project.

We found the method of test development workable in practice. Moreover the approach seemed very manageable, in the sense that we could tell in advance how many test cases we were going to develop and after some time we were able to predict how long a typical test case would take.

The strong point of the method is that there is a precise model from which easily correct traces are derived, leading to a good starting point for editing the final TTCN text of the ATC. The method does not demand that everything is formal or automated, but it is a mix of automated and manuals steps. Therefore it is always possible to insert specific solutions for specific problems not built-in to the models or the tools.

Acknowledgements: the authors want to thank J. Bergstra, P. Boonen, A. Engel, J.F. Groote, R. de Jong, R. Koymans, A. Mikes, J. van de Pol, M. Reniers, O. Sies, J. Springintveld, A. Timmermans, and E. Zondag for the cooperation, help and discussions.

18 REFERENCES

CCITT (1992) Message Sequence Chart (MSC), Rec. Z.120, Study Group X (1992).

ISO (1991) International Organisation for Standardisation, OSI conformance testing methodology and framework, International standard IS 9646.

Knightson K.G. (1993) OSI protocol conformance testing, McGraw-Hill, Inc., 1993.

Mauw S., Veltink G. (1993) Algebraic specification of communication protocols, Cambridge university press, 1993.

Mauw S., Winter T. (1993) A Prototype toolset for interworkings, Philips Telecommunication Review, 51(3), (Dec. 1993), pp. 41–45.

Sidhu D.P., Leung T.K. (1989) Formal methods in protocol testing, a detailed study, IEEE trans. on Softw. Eng., Vol. 15, No. 4, pp. 413–426.

Wehkamp (1996) Wehkamp Homeshopping, interactive world wide web application, at URL `http://www.wehkamp.nl/`.