Chapter 1

FORMAL TEST AUTOMATION: THE CONFERENCE PROTOCOL WITH PHACT

Lex Heerink
Philips Research Laboratories, Eindhoven, The Netherlands
lex.heerink@philips.com

Jan Feenstra and Jan Tretmans*
University of Twente†*, Enschede, The Netherlands*
{feenstra,tretmans}@cs.utwente.nl

Abstract We discuss a case study of automatic test generation and test execution based on formal methods. The case is the Conference Protocol, a simple, chatbox-like protocol, for which (formal) specifications and multiple implementations are publicly available and which is also used in other case study experiments. The tool used for test generation and test execution is PHACT, the PHilips Automated Conformance Tester. The formal method is (Extented) Finite State Machines which is the input language for PHACT. The experiment consists of developing a Finite State Machine specification for the Conference Protocol, generating 82 tests in TTCN with PHACT, and executing these tests against 28 different implementations of the Conference Protocol, both correct and erroneous ones. The result is that some erroneous implementations are not detected by the test cases. These results are analysed, the merits of Extented Finite State Machines for specification are discussed, and the achievements of PHACT are assessed. Moreover, the results are compared with a previous experiment in which the same 28 implementations were tested based on specifications in LOTOS and PROMELA.

Keywords: Conformance testing, case study, formal methods, finite state machines, test generation, test execution.

*Corresponding author: Jan Tretmans, University of Twente, Faculty of Computer Science, Formal Methods & Tools research group, P.O. Box 217, 7500 AE Enschede, The Netherlands; tretmans@cs.utwente.nl.

†This research is supported by the Dutch Technology Foundation STW under project STW TIF.4111: *Côte de Resyste* – COnformance TEsting of REactive SYSTEms; http://fmt.cs.utwente.nl/CdR.

1. INTRODUCTION

In this note we describe a case study of automatic test generation and execution based on formal methods. The case is the *Conference Protocol*, a simple, chatbox-like protocol, for which (formal) specifications and multiple implementations are publicly available. The test tool used is PHACT. The formal method is (Extended) Finite State Machines – (E)FSM – which is the input language for PHACT.

The experiment consists of developing an EFSM specification for the Conference Protocol, generating tests in TTCN with the Conformance Kit – which is a part of PHACT – and executing these tests, also with PHACT, against 28 different implementations, both correct and erroneous ones. The goal of this paper is to present the experiment, to analyse the results, assess the merits of (E)FSM's and of PHACT, and to compare with a previous experiment in which the same 28 implementations were tested. That experiment is described in [1] and this paper is based on it.

2. CONFERENCE PROTOCOL

The Conference Protocol provides a multicast service, i.e., a 'chatbox' service, to its users. A group of users constitutes a conference. Each user in a conference can exchange messages with all other users in that conference. A conference can change dynamically: users can join or leave a conference at any time.

The Conference Protocol is a relatively simple protocol, yet, it contains many aspects of more realistic protocols. It has been used in other case studies, in particular, in [1] we used it with the test tool TORX and formal specifications in LOTOS, PROMELA and SDL. Hence, the Conference Protocol serves as a vehicle for testing and comparison of test tools. The web page [7] provides descriptions of the protocol, its service, formal specifications, and 28 different implementations, one of which is (assumed to be) correct, while the others are mutants in which (small) errors have been introduced.

3. PHACT

PHACT (PHilips Automated Conformance Tester) is a set of tools that has been used within Philips for some time to generate tests automatically and to execute these tests against concrete implementations [4, 8]. The tool set consists of two major parts: a *generator* and an *executor*, see Figure 1.1.

The *generator* mainly consists of the Conformance Kit developed at KPN Research [2, 5]. It takes a formal specification in Extended Finite State Machine format (EFSM, Mealy Machine with state variables) as its input. This EFSM must be *input complete* and *deterministic*, i.e., for every state and input

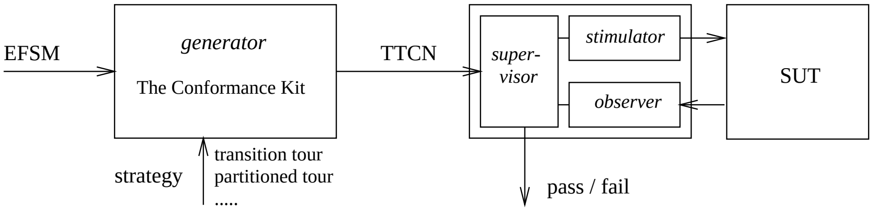

Figure 1.1 PHACT

there shall be exactly one transition that specifies the corresponding output and destination state. The first step of the *generator* is to transform this EFSM into an equivalent Finite State Machine (FSM) by expanding all (necessarily finite-domain) variables. This FSM is minimized into an equivalent minimal FSM, and then tests are generated according to one of the standard, FSM-based algorithms (random sequences, transition tour, partitioned tour, i.e., UIO-based test sequences; see [6] for an overview). The tests are expressed in a subset of TTCN.

The *executor* part provides an environment for the execution of tests against a *System Under Test* (SUT). It consists of a *supervisor*, a *stimulator* and an *observer*. The *stimulator* and *observer* are application specific and must be developed, from a template, for each SUT separately. They provide the functionality for encoding and stimulating the SUT with a single input test event and for observing and decoding an output test event from the SUT. The *supervisor* is a generic kind of test engine. It reads the generated test cases and performs actions and test events given in the test case by using the *stimulator* and *observer* functions.

4. TEST ARCHITECTURE

The test architecture for testing Conference Protocol implementations is the same as the one used in [1], see Figure 1.2. One Conference Protocol Entity (CPE) is tested; it is the IUT (*Implementation Under Test*). This CPE has one local conference user, denoted A, and two remote users, denoted B and C. The CPE communicates with its environment at the IAP's (*Implementation Access Points*) which are the CSAP (Conference Service Access Point) and the USAP (UDP Service Access Point). The test context is formed by the underlying layer, which is assumed to be a reliable and order-preserving UDP layer, sockets used for UDP communication, and pipes for communication at the CSAP. The SUT consists of the IUT and the test context. The PCO's (*Points of Control and Observation*) are the CSAP, which is the Upper Tester PCO (UT) and the remote USAP's which constitute the Lower Tester PCO (LT). The tester is instantiated with the PHACT *executor* (see Section 3.). The *stimulator* and *observer* provide the inputs and observe the outputs, respectively, at the UT and LT PCO's.

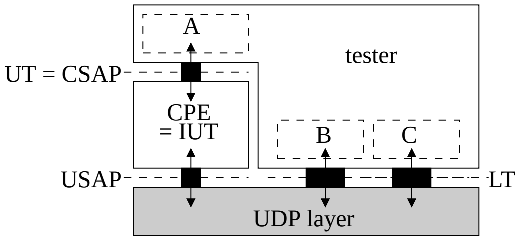

Figure 1.2 Test architecture

5. FINITE STATE MACHINE SPECIFICATION

The first step in the test experiment was developing an EFSM specification of the Conference Protocol. Starting point was the description in [7]. This resulted in an EFSM for one CPE with two conferences, one local user at the upper interface, and two remote users. It is written in the EFSM format of the Conformance Kit and it has 2 states, 3 variables, 18 inputs, 17 outputs and 36 explicitly specified transitions. The FSM expanded from this EFSM contains 9 states.

The restrictions on the (E)FSM imposed by the tool have implications for the way the SUT was modelled. We discuss a few aspects.

In EFSM's there is for every input at most one output. The Conference Protocol has multicast aspects. When user A in Figure 1.2 wants to join a conference, initiated by a *join-SP* (*join-Service-Primitive*) at CSAP, the CPE has to send a *join-PDU* (*join-Protocol-Data-Unit*) to all potential conference partners, i.e., to B and C. So, the input *join-SP* leads to two outputs: a *join-PDU* to B and one to C. Moreover, these outputs may arrive in any order – nondeterministically – at B and C. Neither two outputs in one transition, nor nondeterminism can be modelled directly in EFSM's. We solved this by using a single output action *join-A-to-BC-PDU*, which models the two SUT outputs occurring in any order (cf. [5]).

When the action *join-A-to-BC-PDU* occurs in a test case the *observer* has to take care to implement this action correctly, i.e., to observe two PDU's in arbitrary order. This is implemented by starting a timer before every observation. As long as this timer runs the implementation may perform its outputs. On expiration the *observer* reads all outputs from all PCO's, interprets them together, and tries to map them onto the corresponding single EFSM output action. So, any observation of outputs lasts at least the time required for this timer to expire. Note that this way of observation does not correspond with the standard TTCN snapshot semantics. A consequence of this strategy is that some orderings of actions are not tested, e.g., in the example above, an input action between the two *join-PDU*'s will never be considered for testing.

Alternation between inputs and outputs is always required in EFSM's. This also implies that a sequence of multiple inputs with delayed outputs is not considered, and hence not tested. In fact, EFSM's assume a kind of *synchrony hypothesis*.

Compositionality is not straightforward in EFSM's. This involves the test context. In our previous experiments with LOTOS and PROMELA we could easily combine the CPE model with the context models. For our EFSM model this turned out to be more complex. Actually, we do not take it into account at all, so we make the assumption that the test context, i.e., the queues which model the CSAP pipes and the lower layer UDP, does not influence the observable behaviour of the IUT. Whereas for queues this might seem a reasonable assumption it causes problems if the reliability and ordering assumptions on UDP are released, i.e., if UDP has to be modelled as a lossy bag.

The last important restriction of EFSM's in the Conformance Kit and PHACT is that they do not allow data parameters in inputs and outputs. Whereas in LOTOS and PROMELA different conferences, users and UDP addresses were easily modelled with data parameters, in EFSM's these have to be coded explicitly. This means that simplifications and abstractions have to be made with respect to the data aspects which will be tested in order to avoid an enormous set of explicitly coded input and output actions. Consequently, for all abstracted data aspects there will be no test cases generated. As an example, the input action *join-B-to-A-1-PDU* represents a *join-PDU* with parameters B as source address, A as destination address, and 1 as conference identifier. This explains the large number of actions necessary in our EFSM model.

6. THE TEST EXPERIMENT

With the Conformance Kit 82 tests were generated from the EFSM specification with the partitioned tour method. For each explicitly specified transition of the FSM, obtained from the EFSM, a test case was generated; for transitions only added to make the EFSM input complete, no test cases were generated. A test case following the partitioned tour method consists of (*i*) a *synchronizing sequence* that brings the system from any arbitrary state into its initial state; (*ii*) a *transferring sequence* that brings the system from the initial state to the source state of the transition to be tested; (*iii*) a *transition test*, which performs the input and checks the output of the transition to be tested; and (*iv*) a *Simple Input/Output Sequence* (also known as Unique Input/Output (UIO) sequence), which verifies whether the correct destination state is reached. The length of the test cases varies from 6 to 16 test events: the synchronizing sequence has minimum 2 and maximum 4 test events, the 9 transferring sequences have a length from 0 to 6, every transition test consists of 2 test events, and the 9 UIO sequences have a length from 2 to 4 test events (not counting *start-timer* events).

The 82 test cases were successively applied, using the *executor* part of PHACT, to the 28 Conference Protocol implementations. They were executed for each implementation without resetting and re-initializing the implementation between the different test cases, which is possible due to the synchronizing sequence.

The results of the test experiments are that 21 obtained the verdict *fail*. These 21 implementations were correctly detected: they are known to be erroneous mutants. To 6 implementations the verdict *pass* was assigned, among which there was the correct one. Moreover, the following (erroneous) mutants got a *pass* verdict (using the mutant numbers of our internal identification scheme): 289, 293, 398, 444, and 666. One implementation (mutant 749) led to an abnormal termination (‘core dump’).

In our previous experiment with the Conference Protocol we tested the same 28 implementations with the test tool TORX [1]. Those experiments led to the detection of 25 out of the 28 implementations: the correct implementation and the mutants 444 and 666 got the verdict *pass*.

7. ANALYSIS

While analysing the test results, the first observation was that some of the mutants were only detected because the 82 test cases were executed consecutively without resetting the implementation, i.e., the 82 test cases were actually executed as one large concatenated test case. This led to the situation where an error is triggered in one test case, without causing an incorrect observation, while the subsequently executed test case then led to the failure in terms of an incorrect observation. This situation becomes apparent, for instance, in the mutants numbered 332 and 345. These mutants implement the *leave-PDU* incorrectly, so that a conference partner which leaves the conference is not removed from the implementation’s internal set of current conference partners. When one test case is finished and the synchronizing sequence of the next test case has been executed, the specification FSM is in its initial state again. But, since the SUT is not re-initialized between the two test cases, the SUT is, erroneously, not in its initial state: the conference partner that left is still in the set of current conference partners. This situation then leads to an incorrect observation with *fail* verdict in the subsequent test case, e.g., when PDU’s are sent to the partners in the set.

The second point of analysis concerns mutant 749 that led to an abnormal termination. Analysis shows that this mutant generates an invalid PDU which is not correctly recognized by the *observer* and, consequently, cannot be mapped onto an EFSM output action. It turns out that our developed *observer* of Phact is not robust for unrecognized PDU's, which is an anomaly of our *observer*. Mutant 749 was detected by TorX since the observer part of TorX maps every message which it cannot recognize onto a non-existing PDU. Since, of course, such a PDU does not occur in the specification this is subsequently considered as an incorrect response leading to the verdict *fail*. An analogous solution could have been implemented for Phact.

The third point of analysis concerns the mutants not detected by Phact.

Mutants 444 and 666: Analysis in [1] showed that these mutants accept PDU's from any source, not only from (potential) partners. In the LOTOS and Promela specifications the responses to such PDU's are not explicitly specified, which means that, following TorX' correctness criterion **ioco** [1], any response from the SUT is allowed. Consequently, 444 and 666 are **ioco**-correct and pass TorX.

With Phact we only generated test cases for explicitly specified transitions and not for those that were only added to make the FSM input complete (Section 6.). Consequently, it is no surprise that also with Phact the mutants 444 and 666 were not detected. However, whereas with TorX these mutants can principally not be detected – the necessary stimuli are never generated by TorX – they might have been detected with Phact – they can occur in a synchronizing sequence or UIO sequence. Moreover, in order to minimize the number of data parameters (which have to be coded explicitly; see Section 5.), inputs from non-existent partners were not considered in our EFSM. Hence, such inputs are outside the considered input set and, like for TorX, the necessary stimuli to trigger the faults are never generated.

Mutants 289, 293 and 398: These erroneous mutants got the verdict *pass*, whereas they were detected in the TorX experiment. We now try to analyse why Phact does not detect them by comparison with the TorX experiment. The failure trace-logs of TorX, i.e., the traces of inputs and outputs leading to the detection by TorX, were analysed. Then we projected these failure traces onto the EFSM and analysed the relevant EFSM behaviour and the Phact test suite with respect to these failure traces.

Mutant 293: Mutant 293 uses a bag instead of a set to administer its conference partners. With TorX the following failure trace detects this mutant.

1	*input:*	*UT ! A ! join-SP(A,conf-1)*	*5*	*input:*	*LT ! C ! answer-PDU(C,conf-1)*
2	*output:*	*LT ! C ! join-PDU(A,conf-1)*	*6*	*input:*	*UT ! A ! datareq-SP(m1)*
3	*output:*	*LT ! B ! join-PDU(A,conf-1)*	*7*	*output:*	*LT ! C ! data-PDU(A,m1)*
4	*input:*	*LT ! C ! answer-PDU(C,conf-1)*	*8*	*output:*	*LT ! C ! data-PDU(A,m1) : fail*

In step 1 the local user A wants to start a conference with name *conf-1* via a *join-Service-Primitive* at the upper test interface UT. In steps 2 and 3 the SUT informs the potential partners B and C at the lower test interface LT by sending two *join-PDU*'s. Then, at LT, user C sends an *answer-PDU* to the

SUT, twice. The SUT should react by simply neglecting the second *answer-PDU*, however, the SUT does store the second one in its bag. In step 6 the SUT receives a request from its local user A to send message *m1*. Then, in steps 7 and 8 the SUT sends the message to user C, also twice, which is incorrect: the SUT should send it only once. So, in step 8 no output was expected according to the specification ('quiescence'): *fail*.

Analysis of this sequence of events in the EFSM specification gives:

Steps	*Condition*	*Input*	*Output*	*Action*	*Dest.*
(1-3)	*TRUE*	*join-A-1-PDU*	*join-A-to-BC-1-PDU*	*(A-conf:=1)*	*Conn*
(4)	*(A-conf=1)*	*answer-C-to-A-1-PDU*	*(none)*	*(C-part:=TRUE)*	*Conn*
(5)	*(A-conf=1)*	*answer-C-to-A-1-PDU*	*(none)*	*(C-part:=TRUE)*	*Conn*
(6,7)	*notbc*	*datareq-SP*	*data-A-to-C-PDU*	*(none)*	*Conn*

notbc == (NOT(B-part) AND C-part)

In the first EFSM transition, corresponding to the steps *(1–3)* of the TorX trace, user A joins conference 1 and sends two *join-PDU*'s to users B and C, respectively. (This is modelled as one output, see Section 5.). The EFSM goes to state *Conn* and conference 1 is active: *(A-conf:=1)*. We see in steps *(4)* and *(5)* that user C is allowed to send an *answer-PDU*, twice. The second one does not change the state of the EFSM and, consequently, should have no effect. If in *(6,7)* the local user A issues a *datareq-SP* the corresponding *data-PDU* is sent only once. This state of the EFSM is never reached by the mutant: after transition *(5)* the mutant goes to a state where the set (bag) of current conference partners contains two entries for partner C. This state does not exist in the EFSM specification. Using the partioned tour method no test is generated for such a state and, consequently, the error is not detected. Typically, errors in implementation states which do not have a corresponding state in the specification are not always found by the UIO-based partitioned tour method.

Mutant 289: Mutant 289 does not update its internal set of current conference partners correctly when an *answer-PDU* is received. With an analogous, although somewhat more complex analysis as for mutant 293 we concluded that also this failure of detection was caused by an additional state in the implementation which did not have a corresponding state in the EFSM specification.

Mutant 398: Mutant 398 does not check the conference identifier when partners send *answer-PDU*'s. With TorX we constructed the following failure trace.

1	*input:*	*UT ! A ! join-SP(A,conf-1)*	*4*	*input:*	*LT ! C ! answer-PDU(C,conf-2)*
2	*output:*	*LT ! C ! join-PDU(A,conf-1)*	*5*	*input:*	*LT ! C ! data-PDU(C,m1)*
3	*output:*	*LT ! B ! join-PDU(A,conf-1)*	*6*	*output:*	*UT ! A ! dataind-SP(C,m1) : fail*

User A joins conference *conf-1* in steps 1-3. Then user C sends an *answer-PDU* to the SUT to join another conference: *conf-2*. The SUT should ignore this, but it does not: it erroneously adds C to its set of current conference partners. Then, when it receives in step 5 a *data-PDU* with message *m1* from C, the SUT passes the message *m1* erroneously as *dataind-SP* to local user A.

Transposing this failure trace to the EFSM we obtain the following.

Steps	*Condition*	*Input*	*Output*	*Action*	*Dest.*
(1-3)	*TRUE*	*join-A-1-SP*	*join-A-to-BC-1-PDU*	*(A-conf:=1)*	*Conn*
(4)	*(A-conf=1)*	*answer-C-to-A-2-PDU*	*(none)*	*(none)*	*Conn*
(5,6)	*c1notbnotc*	*data-C-to-A-PDU*	*join-A-to-C-1-PDU*	*(none)*	*Conn*

c1notbnotc == ((A-conf=1) AND NOT(B-part) AND NOT(C-part))

Transition *(4)* is a transition which is added in the EFSM only to make the EFSM input complete. The EFSM specifies simply to neglect the incoming *answer-C-to-A-2-PDU*. Since for such transitions no test cases are derived the mutant is not detected. Probably, mutant 398 would have been detected if test cases had been generated for all transitions including the ones intended to make the EFSM complete, which is a possibility in PHACT. After all, the mutant makes an erroneous transition to a state known in the specification, i.e., the state in which C is a partner in the set of current conference partners. We did not perform this additional experiment yet.

8. CONCLUDING REMARKS

If the Conference Protocol experiments were a match between TORX and PHACT the result would be $25 : 21$ in favour of TORX. PHACT did not detect 4 mutants which TORX did. One of these caused a ‘core dump’; a simple improvement in the *observer* of PHACT would detect this mutant. Another mutant was not detected because we only tested the explicitly specified transitions. A PHACT test suite which tests all transitions would probably be able to detect this one. What remains are two implementations which are not detected because they clearly have states which do not exist in the EFSM specification. Such non-detected errors are typical for the partitioned tour method which is used by PHACT. In addition, we should note that some of the mutants were only detected since all 82 test cases were executed successively without resetting the implementations, see Section 7..

With respect to PHACT, the test tool is usable and successfully detected most of the faulty implementations. Most of its disadvantages are related to the restrictions imposed on the EFSM’s it uses. They were discussed in Section 5.: the required alternation between single input and output actions, determinism, lack of compositionality, and the inability to cope with data parameters.

The Conference Protocol turned out to be a nice, useful and interesting case study: it is simple enough to be understood but not so simple that it is trivial, multiple specifications and implementations are publicly available, and the number of experiments with it increases so that interesting comparisons are possible [3]. As far as we know this is the first real comparison, based on actually detected erroneous implementations, of an FSM-based technique – PHACT – with an LTS-based technique – TORX.

Many additional experiments can be envisaged. First, the experiments mentioned above to detect the two not yet detected mutants with PHACT can be performed. Second, other orderings of test cases or independent test case execution could be considered. Third, other strategies of PHACT can be used: random sequences, transition tours. Fourth, more, and more tricky erroneous implementations can be developed to extend the comparison. Fifth, other test tools can be applied to the Conference Protocol and other case studies can be used to compare PHACT and TORX. Finally, apart from counting detected mutants, other comparison criteria should be investigated, such as total effort and cost of testing, cost per detected mutant, and ease of use.

Acknowledgements The authors would like to thank the members of *Côte de Resyste*, in particular Axel Belinfante, René de Vries and Ron Koymans, and the anonymous reviewers for their comments and support. Erik Kwast from KPN Research is acknowledged for supplying the Conformance Kit and developing the first version of the Conference Protocol EFSM.

References

[1] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal Test Automation: A Simple Experiment. In G. Csopaki et al. (eds.), *Testing of Communicating Systems 12*, pp. 179–196. Kluwer Ac. Publ., 1999.

[2] S.P. van de Burgt, J. Kroon, E. Kwast, and H.J. Wilts. The RNL Conformance Kit. In J. de Meer et al. (eds.), *Protocol Test Systems 2*, pp. 279–294. North-Holland, 1990.

[3] L. Du Bousquet, S. Ramangalshy, C. Viho, A. Belinfante, and R.G. de Vries. Formal Test Automation: The Conference Protocol with TGV/TORX. In *TestCom 2000*. Kluwer Ac. Publ., 2000. This issue.

[4] L.M.G. Feijs, F.A.C. Meijs, J.R. Moonen, and J.J. Wamel. Conformance Testing of a Multimedia System using PHACT. In A. Petrenko and N. Yevtushenko (eds.), *Testing of Communicating Systems 11*, pp. 193–210. Kluwer Ac. Publ., 1998.

[5] E. Kwast, H. Wilts, H. Kloosterman, and J. Kroon. User Manual of the Conformance Kit. Version 2.2, PTT Research Neher Labs, Leidschendam, The Netherlands, Oct. 23 1991.

[6] D. Lee and M. Yannakakis. Principles and Methods for Testing Finite State Machines – A Survey. *The Procs. of the IEEE*, 84, Aug. 1996.

[7] Project Consortium Côte de Resyste. Conference Protocol Case Study.
URL: `http://fmt.cs.utwente.nl/ConfCase`.

[8] T.I.P. Trew, B. Lanaspre, M. Hollenberg, J. Springintveld, and T.J. Harosia. Delivering High Definition TV to the USA – Testing Subcontracted Embedded Real-Time Software. In 16^{th} *Conf. on Testing Computer Software (TCS'99)*, Washington D.C., June 1999.