14

FORMAL TEST AUTOMATION: THE CONFERENCE PROTOCOL WITH TGV/TORX

Lydie Du Bousquet, Solofo Ramangalahy, Séverine Simon, César Viho[†]
IRISA, Campus de Beaulieu, F35042 Rennes, France
{ ldubousq, solofo, ssimon, viho } @irisa.fr

Axel Belinfante, René G. de Vries
Department of Computer Science – University of Twente
P.O. box 217, 7500 AE Enschede, the Netherlands
{ belinfan, rdevries } @cs.utwente.nl

Abstract We present an experiment of automated formal conformance testing of the Conference Protocol Entity as reported in [2]. Our approach differs from other experiments, since it investigates the combination of the tools TGV for abstract test generation and TorX for test execution.

Keywords: Conformance testing, test generation and execution, TGV, TorX

1. INTRODUCTION

As conformance testing theories and tools mature (i.e., leading to standards [5, 6]), it is interesting to compare different approaches by practical experiments. In [2] a comparison among several tools, specification formalisms and test execution paradigms was made for an implementation of a simple chat box protocol—called the *conference protocol*. In this paper we describe a similar

[*] This research was partly supported by the NWO in Van Gogh program for French-Dutch cooperation, project "Automatic Generation of Conformance tests", VGP 62-480 - 99.031 and by the Dutch Technology Foundation STW under project STW TIF.4111: *Côte de Resyste* – COnformance TEsting of REactive SYSTEms.

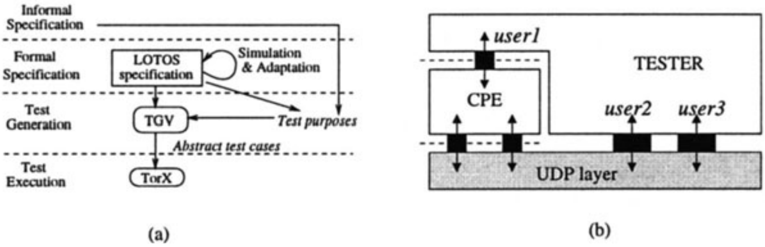
[†] First and third author with INRIA, second with DYADE/INRIA, fourth with IFSIC/Université de Rennes I

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35516-0_20

H. Ural et al. (eds.), *Testing of Communicating Systems*

study using the TGV test generation tool [7], and the TORX tool environment [2] for test execution (see Fig. 1.(a)).

We use two different methods to generate the test suites: one based on manually-designed test purposes, and the other based on randomly-generated test purposes. Our goal is to evaluate advantages and drawbacks of the tools used during this experiment, and to investigate these two approaches of test purpose design.



(a)

(b)

Figure 1. Architecture: (a) steps and tools of the experiments; (b) the test architecture

2. THE EXPERIMENT

2.1 Description

The experiment addresses testing of a multicast protocol implementation, facilitating a service similar to a `chatbox' to users participating in a conference. A conference is a session in which a group of users can participate by exchanging messages with other users (called *partners*). The partners involved in a conference can change dynamically. The Implementation Under Test (IUT) is the *Conference Protocol Entity* (CPE) which implements the protocol at a conference user site. The test architecture of the experiment is depicted in Fig. 1.(b). Since the IUT communicates using the User Datagram Protocol (UDP) underlying service, we need to incorporate it into the System Under Test (SUT). The specification of this protocol is described using the formal language LOTOS.

The experiment consists of testing 28 different implementations of the CPE—of which 27 are incorrect—and detecting the incorrect implementations (i.e., mutants [11]). In each mutant a single error has been introduced. There are three types of errors: *No outputs*, *No internal checks* and *No internal updates* (see [2]). There are respectively 6, 4, and 17 mutants for each type.

We evaluate the quality of the generated test suites by checking how many of these mutants are declared non-conforming with respect to the specification. See [2, 3] for more details about the experiment, the protocol and specifications.

2.2 Test Suite Generation

Description of TGV. TGV [7] is a tool dedicated to the automatic generation of conformance tests based on a formal specification. Given a formal specification of a system to be tested and a formal description of a test purpose, TGV generates an abstract test case. The test purposes are used as test selection criteria and are formalized using automata. An abstract test case is a directed graph in which each path represents a test sequence with associated verdicts which indicate whether the SUT conforms to the specification. We use the conformance relation **ioco** [12], which is a correction notion for conformance between the specification and an implementation. The main characteristic of TGV is that it produces test cases "on-the-fly", i.e. the generation is done in a "lazy" way, so that the specification state space is not completely stored. "On-the-fly" techniques are one of the solutions proposed for the state-space explosion problem commonly encountered in verification techniques.

Formal specification. We used the LOTOS specification, which is freely provided by the Côte de Resyste project [3]. As shown in Fig. 1.(b), the core of the LOTOS specification is a (state-oriented) description of the CPE behaviour. The CPE behaviour is parameterized with the potential conference partners. The instantiation of the CPE with concrete values for these parameters is part of the specification. We made some small modifications on the specification, to make processing by TGV more efficient. These adaptations are similar to optimizations of the PROMELA specification reported in [2].

Test purpose design. In addition to a specification, we need test purposes [5]. In TGV, test purposes are given in the form of an automaton. They serve as a "guide" to the state-space exploration which is performed on the product between the specification and the test purpose. We followed two approaches to obtain them:

- In the first approach, the test purposes are designed manually, based on the informal requirements of the conference protocol [3].
- The second approach consists in automatic random generation of test purposes.

We began by designing 18 basic test purposes for basic protocol functionalities: joining and leaving the conference and data transfer. From these, we composed 8 more complex test purposes. We designed the test purposes to fulfill the informal requirements for a CPE. With the 8 complex test purposes and 11 basic test purposes, we generated a test suite with TGV. We select only 11 of the 18 basic test purposes to produce (basic) test cases with TGV because of the following testing equivalence hypothesis. Since the specification indicates that *user2* and *user3* behave equivalently towards *user1* (see Fig. 1.(b)), the verdict for a test case with *user1* and *user2* (noted *TC(user1,user2)*) should be the same as the verdict for *TC(user1,user3)*, i.e., the same test case replacing *user2* by *user3*. The time effort spent on designing and writing these 19 test purposes and generating tests was 4 hours. After execution of the test suite as described above, we designed 7 new test purposes, since the generated test suite was not able to detect one last mutant. To find the last mutant, we relied on the fault model used for the mutant generation. For each expected error, we designed a specific test purpose for which an implementation with this error would behave incorrectly. Those test purposes were more difficult to design than the previous ones. We decided to stop the test purpose design after ten hours of work.

For the second approach, we used the CADP [4] simulator to simulate the specification randomly. With this tool, we produced and saved 200 traces of 200 steps. We translated those traces into test purposes (with a script), and we used TGV to produce the associated test cases.

2.3 Test Execution

TorX and adaptation. For the execution of the tests generated by TGV we used TORX [2] (configured for on-the-fly test generation and execution) and gave it the tests generated by TGV as "specifications". Because TORX can handle nondeterministic graphs, we were able to execute not only tests cases, but also uncontrollable tests graphs [7]. With TORX, we could reuse ADAPTER, the component that connects the tester to the SUT, as it was configured for the previous conference protocol experiments [2]. Minor changes were necessary for the components PRIMER, which implements the test derivation algorithm, DRIVER, which controls the testing, and EXPLORER, which explores the transition graph. PRIMER had to be changed because it initially only implemented the **ioco** test generation algorithm, while for this experiment it has to compute the traces of its input, i.e., test cases in TGV output format. In addition, it has to recognize the special events in the tests that encode quiescence [12] and verdicts and pass the verdicts to the DRIVER. The DRIVER had to be extended to accept verdicts from the PRIMER (in addition to computing the verdicts itself). The CADP libraries [4] allowed us to replace the LOTOS-specific EXPLORER by one that read the test case in BCG format.

Results. Since TGV is based on the **ioco**-conformance relation, it can potentially detect **ioco**-incorrect implementations, but it cannot detect **ioco**-correct erroneous implementations. Among the 28 implementations, 25 were **ioco**-incorrect.

We first executed the test cases generated from the manually-written test purposes. In total 34 test cases were derived from 19 test purposes (11 basic

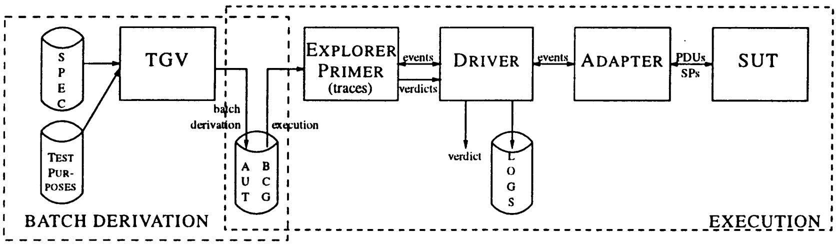


Figure 2. TORX tool architecture instantiated with TGV

and 8 complex). The test cases detected 24 **ioco**-incorrect mutants. One mutant was detected by all the test cases, 6 mutants were detected by 50% of the test cases, 7 mutants were detected by less than 10% of the test cases. The test suite generated from the 7 test purposes especially designed to find the last undetected mutant, was not able to detect it.

We then executed the test cases produced from the randomly generated test purposes. All 25 **ioco**-incorrect mutants were detected. One was detected by all the test cases, 3 were detected by 50% of the test cases, 3 were detected by less than 10% of the test cases.

3. CONCLUSION AND PERSPECTIVES

Tools. We were able to easily interface TGV and TORX, so that we could execute the generated tests. As a result of this experiment, we noted two ideas for the improvement of TGV, both of which have been implemented. First, the restriction that a test purpose should be deterministic has been removed, to facilitate the writing of a test purpose. For example, it eases the writing of wild cards in the transitions of a test purpose by avoiding the manual computation of the complement of a regular expression. The other adaptation of TGV addresses the controllability of a generated test case. In TGV, priority between inputs and outputs (of the specification) was given to the inputs. It can be advantageous to give less control to the environment (e.g., regarding quiescence [12]).

Specification of test purposes. Manually-written test purposes were more general than those produced by the random generation method: wild cards in transitions labels were used whereas they did not appear in random tests purpose coming from traces. To derive a test case by TGV, we began with a general test purpose and subsequently, restricted it (i.e., instantiated some wild cards). This iterative approach was needed, since the generation either took too much time or TGV did not produce a test case because of incoherence between the specification and the test purpose. That is, a test purpose is coherent with a specification if the behaviors it describes are included in those of the specification, see [7].

This experiment with TGV has underlined the difficulties that users have when translating a test purpose into TGV input format with which test cases are quickly generated. A balance has to be found between the expressiveness of test purposes and the amount of computation needed to generate a test case. This expressiveness comes from the ability to use wild cards in the labels. When test case generation takes too long, the expressiveness of the test purpose has to be reduced (e.g., by expanding the wild cards labels or by considering the different options of TGV). To succeed in a test campaign with manually-written test purposes, the user has to know the algorithms described in [7] to understand how TGV works in order to know what to do to obtain the expected test case.

From the experience gained in the use of TGV with the manually-written test purposes we chose to have the random test purposes as constrained as possible to ensure fast generation of test cases. This shows that it is possible to choose the level of automation of test generation: completely automated (as with random test purpose) or with TestComposer [8] or in a more interactive way (more control over the test generation process).

Manual vs. random test purpose design. One should notice that the goal of this paper is not to compare the efficiency of the two test purpose design approaches. The fact that one method is better than another one cannot be established with a single experiment. We present our observations on the two design approaches. With the random generation of test purposes approach, all ioco-incorrect mutants were found. It is not clear why we did not manage to detect one ioco-incorrect mutant with the manual approach. The fact that all the mutants were detected by the random approach is not surprising considering that this method is essentially equivalent to the on-the-fly method of [2]. The efficiency of the random walk method for protocol (specification) testing has been known for a long time [13] even though few formal explanations exist regarding its efficiency [10]. Efficiency is transferred from specification testing to conformance testing when execution is combined with simulation. Another reason for the efficiency of both methods could be that the fault model we considered is restricted. Only "functional" mutants (in contrast to the usual syntactic mutants) were considered. The high-level fault model (no outputs, no internal checks, no internal updates), when applied at the source-code level of the implementation, give less choice in mutations than usual mutation operators [1] applied directly on the code of the implementation.

Analysis. As in the previous experiment [2], the analysis of test case execution was the least automated part. We did not go much further than verdict checking, i.e., we did not diagnose the errors. We found that analysis of random-generated test purposes is quite difficult. In fact, many events in the test execution trace were unnecessary to trigger the error. The analysis of the trace produced from designed test purposes seems easier since these test purposes usually give a more precise idea of what is supposed to be tested (although obtaining a fail verdict is not necessary related to a fail on the "property" targeted by the test purpose). To diagnose a fail verdict on a mutant, we used an iterative approach to analyze the result of the test case execution, i.e., analyzing the test run. From this test run, we made a new test purpose by suppressing irrelevant events and derived a new test case. After execution of this test case, we iterated again to converge to a "minimal" test purpose. From a reduced test purpose, we hoped to interpret the detected error more easily. Although this methodological approach eases human diagnosis, it is not systematic and did not help for the last undetected mutant with manual test purposes[1]. Here, automation is lacking for fault diagnosis (which is a more complex problem than conformance testing, see [9] p. 1119).

Comparison with previous experiments. Both approaches for test purpose development, resulted in a good mutation score (96% and 100% respectively). These results should be somewhat mitigated by the fact that mutant population is low and maybe not representative of common errors encountered when writing C code. We consider this experiment to be a case study. It is not possible to really compare this experiment with a previous one, i.e., benchmark it. The only conclusion to be drawn is that TGV and TORX (the full tool, not just the ADAPTER) have the same fault detecting power on this case study, which is not surprising since they use the same conformance relation. To do real benchmarking you need, besides benchmarking criteria, equivalent specifications in order to make a fair comparison. On the other hand, specifications may be adapted so that a tool is able to perform better. Therefore strict equivalence among specifications is hard to maintain.

Perspectives. The experiment detailed here is the first part of a larger one using other tools. We intend to use TGV and TestComposer with the SDL specification of the CPE, and we have already produced some test cases. Currently we are working on the execution of these test cases. We want to extend the set of mutants by automatically generating more mutants. We are currently studying possible approaches towards this. We believe that more work is needed on the notion of a test purpose both at the theoretical and method-

[1] Note that since we obtained traces leading to **fail** with random test purposes, we could have used them for the other approach. But since we did not get the "meaning" of these test purposes, they cannot be considered as manual test purposes.

ological level. There are several informal definitions of what a test purpose is. TGV uses one specific, formal and constructive notion of a test purpose. One can consider that the test purpose description in TGV is too restrictive (for instance, it is not possible to specify quiescence in test purposes). It remains an open problem, to find the correct notion of test purpose. Such a definition should be as general as the informal one in [5] or the formal definition in [6]. In addition, we would still like to use the definition as a basis for test generation. Finally, tool support for writing test purposes is needed.

References

[1] H. Agrawal, R. A. DeMillo, B. Hataway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, and E. Spafford. Design of mutant operators for the C programming language. Technical Report TR-41-P, SERC, 1989.

[2] 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 *IWTCS'99*, pages 179–196. Kluwer, 1999.

[3] Côte de Resyste. Conference protocol case study. http://fmt.cs.utwente.nl/ConfCase, 1999.

[4] H. Garavel. Open/cæsar: An open software architecture for verification, simulation, and testing. In *TACAS'98*, LNCS 1384. Springer, 1998.

[5] ISO. *International Standard IS-9646.* 1991.

[6] ITU-T recommendation Z-500: Framework on formal methods in conformance testing, 1997.

[7] T. Jéron and P. Morel. Test generation derived from model-checking. In *CAV'99*, LNCS 1633. Springer, 1999.

[8] A. Kerbrat, T. Jéron, and R. Groz. Automated test generation from SDL specifications. In *SDL'99*, pages 135–151. Elsevier, 1999.

[9] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines–a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[10] M. Mihail and C. H. Papadimitriou. On the random walk method for protocol testing. In *CAV'94*, LNCS 818, pages 132–141. Springer, 1994.

[11] R. J. Lipton R. A. DeMillo and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, 1978.

[12] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[13] C. H. West. Protocol validation by random state exploration. In *PSTV'86*, pages 7.1–7.12, 1986.