Chapter 1

FORMAL TEST AUTOMATION: A SIMPLE EXPERIMENT

Axel Belinfante*, Jan Feenstra, René G. de Vries, Jan Tretmans
University of Twente

Nicolae Goga, Loe Feijs, Sjouke Mauw
Eindhoven University of Technology

Lex Heerink
Philips Research Laboratories

Abstract In this paper[1] we study the automation of test derivation and execution in the area of conformance testing. The test scenarios are derived from multiple specification languages: LOTOS, PROMELA and SDL. A central theme of this study is the usability of batch-oriented and on-the-fly testing approaches. To facilitate the derivation from multiple formal description techniques and the different test execution approaches, an open, generic environment called TORX is introduced. TORX enables plugging in existing or dedicated tools. We have carried out several experiments in testing a conference protocol, resulting in requirements on automated testing and benchmarking criteria.

*Corresponding author: Axel Belinfante, University of Twente, Faculty of Computer Science, Formal Methods and Tools research group, P.O. Box 217, NL-7500 AE Enschede, The Netherlands, email: Axel.Belinfante@cs.utwente.nl
[1]This research is supported by the Dutch Technology Foundation STW under project STW TIF.4111: *Côte de Resyste* – COnformance TEsting of REactive SYSTEms. URL: http://fmt.cs.utwente.nl/CdR

1 INTRODUCTION

Conformance testing is an important activity in the development of reactive systems. Its aim is to gain confidence in the correctness of the system, by means of experimenting with the system implementation. To judge whether an implementation is a valid realization of the specification, and to compare tests and test results, we need a precise notion of correctness. Using formal methods we can achive this. Another benefit from the use of formal techniques is that it allows to automate the conformance testing process. This is important since test derivation is an error-prone and time-intensive process. In [8] a framework for conformance testing based on formal methods (FMCT) is described. Previously, we showed in [11] the testing of a relatively simple example, the conference protocol, in order to assess the feasibility of the FMCT model. One of the major conclusions was that FMCT provides a sound basis for conformance testing based on formal methods, but automation is necessary.

In this paper, we study this automation of conformance testing, by comparing different approaches of test execution, different formalisms as the basis for test derivation, and different supporting tool sets. The aim is to get insight in the strengths and weaknesses of the different approaches, to identify shortcomings, and to identify comparison criteria, required computational effort and means to accommodate automation. These results are an inspiration for an elaborated study of benchmarking existing automated testing methods and tools. The research reported in this paper is part of the *Côte-de-Resyste* project (a Dutch joint venture of the Universities of Eindhoven and Twente and the industrial partners Philips Research and KPN Research). The project aims, among other goals, at comparing existing automated test methods and developing open testing tools together with the underlying formal theory.

The results of this paper are obtained from the same case study as in [11], i.e. testing the *conference protocol*. The difference is that now we do it automatically. Tests are derived from multiple *formal description techniques* (FDTs): LOTOS, PROMELA and SDL. The "simplicity" of our "experiment" merely refers to this case study, not to the testing theory or tools that we use. The conference protocol is a rather simple, almost toy-like protocol.

In this paper, the conference protocol is automatically tested in two ways: on-the-fly and batch-wise. Our on-the-fly testing experiments are based on LOTOS and PROMELA. The batch testing experiment is based on the TAU tool [1] AUTOLINK which is based on the work in [10], i.e. the derivation of TTCN test cases from SDL in a semi-automated way. Test derivation and test execution are facilitated by *Côte de Resyste* tools available in the TORX environment. This is a tool architecture giving means to link different kinds of test tools within a test derivation/execution site, without reengineering of the whole test site, thus facilitating an open generic environment. For instance, for this study we can plug in several modules for the support of the FDTs. Also TTCN is supported, which can be plugged into the execution part of TORX.

This paper is structured as follows. Section 2 gives an overview of test derivation and test execution methods. The TORX environment will be introduced.

Section 3 will explain the conference protocol case study: informally, the formal specifications in LOTOS, PROMELA and SDL, and the implementations. Section 4 deals with the test architecture for testing conference protocol entity implementations. Section 5 reports about the test activities we have carried out. We give an evaluation of the results and directions for future work in the final section.

2 AUTOMATED TESTING

In system development we build an implementation i based on a specification s. Formally, i is said to be a correct implementation of s if i **imp** s, where **imp** is an implementation relation, i.e. the notion of correctness. An implementation which is assessed on its correctness by testing is called an *Implementation under Test* (*IUT*). During test execution a set of test experiments, called a *test suite*, is carried out on this IUT, resulting in a verdict of either *pass* or *fail*. A test suite is exhaustive if we can conclude from the verdict *pass* that the implementation is correct; it is sound if the verdict *fail* never occurs with a correct implementation. Exhaustive testing, i.e. showing the absence of errors, usually requires an infinite test suite and is therefore not feasible in practice. A minimal requirement on a test suite is that it is sound [8]. The *test derivation* process aims at deriving a test suite from the specification, given **imp**.

It follows from the above discussion that there are two main phases in the testing process: *test derivation*, i.e. obtaining a test suite, and *test execution*, i.e. applying the test suite to the IUT. Both phases can be automated. This can be done in two ways: as two separate phases, or in an integrated manner.

In the first approach, in the first phase a test suite is derived and stored in some representation, usually TTCN. In the second phase this test suite is executed along with the IUT. This principle is called *batch testing*. Batch test derivation is computationally expensive and suffers from the state space explosion problem. This complexity can be reduced by user guidance and on-the-fly derivation techniques [4].

The second approach is called *on-the-fly testing*. As opposed to batch testing, test derivation and test execution occur simultaneously. Instead of deriving a complete *test case* (one test scenario in a test suite), the test derivation process derives *test primitives* from the specification. Test primitives are actions that are immediately executed in the test run. While executing a test case, only the necessary part of a test case is considered: the test case is derived *lazily* (cf. lazy evaluation of functional languages). Using observations during the test execution we can reduce the effort in deriving test information from the specification compared with batch derivation; see also [13].

Now that we have introduced the two test methodologies, batch testing and on-the-fly testing, we make a few remarks about their respective qualities. Firstly, the batch-wise approach is better suited for manual test case preparation and for semi-automatic test case preparation. Humans are good at test selection, but they are not fast enough to do it at run time, except perhaps for very slow protocols. This was also one of the traditional ideas behind

TTCN. But now we are moving towards a further automation of the process, and therefore this advantage of batch-wise approach counts as less significant.

The second remark concerns the system dependent PIXIT software, sometimes called mapping software, glue software, encoding/decoding, or interfacing. For batch-wise test derivation it is possible to compile the abstract test cases into concrete test cases which have all the mapping details encoded. For the on-the-fly approaches the encodings and decodings have to be done by run-time facilities. In this paper we will show that this is feasible.

The third remark is that in case of on-the-fly testing all computations have to be done at run-time, whereas batch-wise testing allows some of the work to be moved to compile-time. So, the batch-wise approach has an advantage which makes it easier to satisfy the IUT's real-time requirements. But the price to pay for this is that many test-steps which do not happen at run-time are pre-computed, just because the system happens to choose another branch. This leads to test-suites of an enormous size, and the amount of pre-computation work and the storage demands involved may well undo the advantage.

Test Tool Architecture A test tool architecture was devised which allows on-the-fly testing, batch test derivation and batch test execution for different specification formalisms. This architecture was baptized TorX. The main characteristics of TorX are its flexibility and openness. Flexibility is obtained by requiring a modular architecture with well-defined interfaces between the components – this allows easy replacement of a component by an improved version, or by a component implementing another specification language or implementation relation. Openness is acquired by choosing, when possible, existing (industry standard) interfaces to link the components – this enables integration of 'third party' components that implement these interfaces, in our tool environment. Later we will show how this general architecture was instantiated for on-the-fly testing with LOTOS and Promela and for batch testing with SDL. We now discuss the TorX architecture (Figure 1.1) in terms of its components, the interfaces between them, and the currently available component implementations.

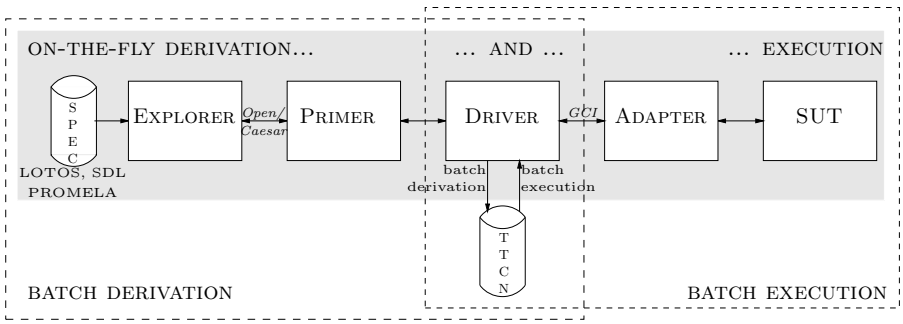


Figure 1.1 TorX tool architecture

TorX consists of the following components (modules): Explorer, Primer, Driver, Adapter, and TTCN storage. Figure 1.1 depicts how these components are linked for batch derivation (with Explorer, Primer Driver and TTCN storage) batch execution (with TTCN storage, Driver, Adapter), and on-the-fly derivation and execution (involving all components without storage of TTCN). The SUT is the *System under Test*. It is the IUT together with its test context, i.e. its surrounding environment (see Section 4).

Explorer. The Explorer is a specification language-specific component that offers functions to explore the transition-graph of a specification and to provide, for a given state, the set of transitions that are enabled in this state. For the interface between Explorer and Primer we use the Open/Caesar interface [6] which is a C API that provides exactly such state and transition functions for labeled transition systems. This implies that we can use existing tools to implement the Explorer. Currently, we use Caesar as our LOTOS Explorer. An SDL explorer also exist (but, to our knowledge, is not publically available).

Primer. The Primer uses the functions provided by the Explorer to implement the test derivation algorithm, for which it keeps track of the set of states that the specification might be in. It offers functions to generate inputs (stimuli) for the implementation and to check outputs (observations) from the implementation. Our current implementation of the Primer implements the test derivation algorithm for the implementation relation **ioco** [12]. Since this algorithm does not contain test data selection criteria, test selection is currently implemented by making random choices using a random number generator. The seed of this generator is a parameter of the Primer.

Driver. The Driver is the central component of the tool architecture; it controls the progress of the testing process. It decides whether to do an input action or to observe and check an output action from the implementation. The Driver uses the Primer to obtain an input and to check whether the output of the implementation is correct. It uses the Adapter to execute the selected inputs by sending these inputs to the IUT, and to observe outputs that are generated by the IUT. For batch testing the derived tests are first stored in a filesystem (indicated as "TTCN" in Figure 1.1). To execute tests in batch mode the test events are obtained from storage rather than from the Primer.

The Primer-Driver interface is only used by the on-the-fly tester, where Primer and Driver are connected by pipes that the Driver uses to write (textual) commands to and receive (textual) responses from the Primer.

Adapter. The Adapter provides the connection with the SUT. It is responsible for sending inputs to and receiving outputs from the SUT on request of the Driver. The Adapter is also responsible for encoding and decoding of abstract actions to concrete bits and bytes, and vice versa. This also involves mapping of the *quiescent action* δ onto time-outs, see [12].

We currently use two interfaces between Driver and Adapter. In our current on-the-fly tester we use a simple (ad hoc) interface based on calling

conventions for the (TCL) functions that implement the en/decoding functions. We are in the process of replacing this interface by the Generic Compiler/Interpreter Interface (GCI) [2] which we already use for batch execution of TTCN test-suites derived by TAU. The GCI has been developed in the INTOOL project for TTCN-based testers as an interface between the TTCN-dependent part (code generated by a TTCN compiler, or a TTCN interpreter) and the other parts (manager, responsible for configuration and logging, and adapter, responsible for encoding and decoding and access to the SUT). The GCI is defined in a language-independent way; a C language binding has been defined, which we use.

Test Approaches in TorX As Figure 1.1 shows, the components can be put together for the three different ways of testing: on-the-fly testing, batch test derivation and batch test execution. We now describe how these configurations were used for testing based on LOTOS, PROMELA and SDL specifications. We will pay special attention to the differences and similarities in the use of the basic building blocks.

On-the-fly testing. Testing for LOTOS and PROMELA is performed on-the-fly based on the implementation relation **ioco** [12]. They use the same DRIVER and major parts of the ADAPTER modules; they differ in the EXPLORER and PRIMER modules.

The DRIVER sends commands to the PRIMER to request a menu of possible input or output events, or the 'execution' of a specific transition from such a menu. Even though it can be configured to fully automatically select a trace to test, the random choices that it makes are 'parameterized' by the seed of its random number, which can be chosen by the user. The test trace that is derived and executed can be logged and replayed during a subsequent test run to guide the DRIVER. The DRIVER is implemented using the scripting language EXPECT because the high-level constructs offered by EXPECT allow rapid prototype development and easy interaction with external programs.

The ADAPTER contains the en/decoding routines and the SUT connection programs, and the function that maps abstract actions onto PCOs (*Points of Control and Observation*, see Section 4). The en/decoding routines and the mapping function are specification language dependent, because they depend on the representation of the abstract values that they have to en/decode, respectively map, which will vary between languages. The SUT connection programs are specification language independent; separate programs are used that handle specific protocols (like TCP, UDP) and can be controlled via standard input and output.

For both LOTOS and PROMELA on-the-fly testing, the mapping of TORX components on tool-implementation programs is the same: the DRIVER forms the main program together with the en/decoding part of the ADAPTER; EXPLORER and PRIMER are integrated in a second program (a.k.a. the *specification module*), and for the SUT connection part of the ADAPTER we use separate protocol-specific *connection programs*.

LOTOS. The specification-dependent EXPLORER module can be automatically generated from a LOTOS specification using the CADP tool set. This EXPLORER module is linked with the **ioco**-PRIMER using the Open/Caesar interface, which gives us a program that has to be configured with lists of input and output gates, and (optionally) with the seed of its random number generator. The PRIMER module is independent of the specification, and, is in principle even specification language independent (for all specification languages for which there is a compiler that compiles to the Open/Caesar interface). In practice this will only be true if the labels generated by the compiler sufficiently resemble LOTOS events (our current PRIMER requires that). The labels generated by the CADP generated EXPLORER do not contain free variables, because CADP expands free variables by enumerating the values in the domains of the variables after which it generates a label for each possible combination of these values.

PROMELA. For PROMELA on-the-fly testing, we can automatically generate a single module that implements both the EXPLORER and PRIMER. This is done by the the TROJKA tool which is described in detail in [13]. The resulting specification module can be configured in the same way as the LOTOS one, but the configuration parameters are different, for example because input and output channel operations are already identified in a PROMELA specification and need not be given by the user. If the tool SPIN were able to supply an Open/Caesar interface for PROMELA, we would have been able to use the same PRIMER as for LOTOS. Such an extension is left for future work.

Batch SDL testing. For SDL we use batch testing to derive and execute test suites in TTCN. For this purpose we use the TAU tool set [1]. The functionality of the EXPLORER, PRIMER and DRIVER is covered by TAU's AUTOLINK test derivation tool [10]. It generates the constraints and dynamic parts of TTCN test suites, guided by *message sequence charts* (MSCs) that have to be provided by the user. These MSCs can be derived by hand from the SDL specification using the SDL simulator that is integrated in TAU. The TTCN test suite derived by AUTOLINK has then to be completed with declarations (e.g. for types and PCOs) using a program that is automatically generated from the SDL specification by TAU's LINK TOOL; the result is a complete TTCN test suite.

The batch test execution DRIVER module is automatically generated from the complete TTCN suites by the TTCN compiler of TAU. This DRIVER is linked with an ADAPTER module using the GCI interface; the result is a single program that can execute the test suites. The ADAPTER differs from the one for the on-the-fly testers in the following aspects. Firstly, the ADAPTER uses the GCI interface, and it does not have to provide a function to map labels to PCOs because the TTCN test suite already explicitly refers to PCOs. Secondly, the ADAPTER does not have to use external programs to provide the connection to the SUT, because support for several connection types (protocols) has been built in. Finally, the en/decoding routines are implemented in C instead of TCL, and they translate using an intermediate representation.

An important difference between the on-the-fly testers used for LOTOS and PROMELA on the one hand, and the batch testing provided using TAU on the other hand, lies in the level of automation offered for test derivation: the on-the-fly testers are able to automatically derive and execute tests without human intervention or guidance, whereas the TAU batch test deriver cannot do its work without a (manually derived) MSC that represents the test purpose.

3 THE CONFERENCE PROTOCOL

The example protocol that is used as the case study for the test experiments is the Conference Protocol [5, 11]. Some aspects of it are highlighted here; an elaborate description together with the complete formal specifications and the set of implementations can be found in [9].

Informal description. The conference service provides a multicast service, resembling a 'chatbox', to users participating in a conference. A conference is a group of users that can exchange messages with all conference partners in that conference. Messages are exchanged using the service primitives *datareq* and *dataind*. The partners in a conference can change dynamically because the conference service allows its users to *join* and *leave* a conference. Different conferences can exist at the same time, but each user can only participate in at most one conference at a time.

The underlying service, used by the conference protocol, is a point-to-point connectionless and unreliable service provided by the *User Datagram Protocol* (UDP), i.e. data packets may get lost or duplicated or be delivered out of sequence but are never corrupted or misdelivered.

The object of our experiments is testing a *Conference Protocol Entity* (CPE). The CPEs send and receive *Protocol Data Units* (PDUs) via the underlying service provide the conference service. The CPE has four PDUs: *join-PDU*, *answer-PDU*, *data-PDU* and *leave-PDU*, which can be sent and received according to a number of rules, of which the details are omitted here. Moreover, every CPE is responsible for the administration of two sets, the *potential conference partners* and the *conference partners*. The first is static and contains all users who are allowed to participate in a conference, and the second is dynamic and contains all conference partners (in the form of names and UDP-addresses) that currently participate in the same conference.

Figure 1.2 gives two example instances of behaviour: in (*a*) a *join* service primitive results in sending a *join-PDU*, which is acknowledged by an *answer-PDU*; in (*b*) a *datareq* service primitive leads to a *data-PDU* being sent to all conference partners, which, in turn, invoke a *dataind* primitive.

Formal specifications. Three formal specifications were developed for the Conference Protocol using LOTOS, PROMELA and SDL.

LOTOS The LOTOS specification was mainly taken from [11]. The core of the specification is a (state-oriented) description of the conference protocol entity behaviour. The CPE behaviour is parameterized with the set of potential conference partners and its CSAP and USAP addresses, and is constrained by

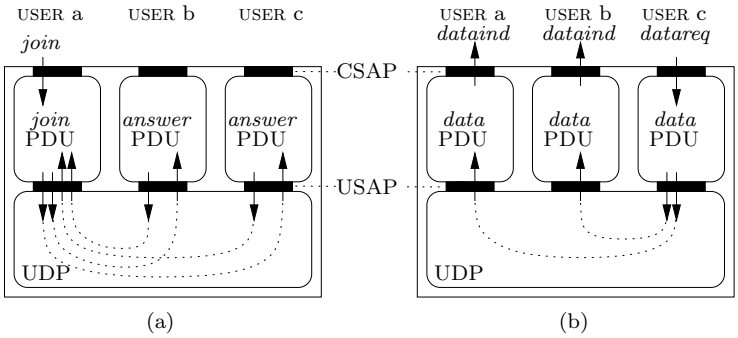


Figure 1.2 The conference protocol

the local behaviour at CSAP and USAP. The instantiation of the CPE with concrete values for these parameters is part of the specification.

PROMELA Communication between conference partners has been modelled by a set of processes, one for each potential receiver, to 'allow' all possible interleavings between the several sendings of multicast PDUs. Instantiating the specification with three potential conference users, a PROMELA model for testing is generated which consists of 122 states and 5 processes. For model checking and simulation purposes, the user needs not only the behaviour of the system itself but also the behaviour of the system environment. For testing this is not required, see [13]. Only some channels have to be marked as observable, viz. the ones where observable actions may occur.

SDL For SDL, several specifications of the conference protocol were produced. In the TAU tool set it is possible to use different kinds of test formalisms, e.g., interoperability and conformance testing. For each of these formalisms different specifications are required. The conference protocol model is specified in a natural way by decomposing the problem into three subprocesses: one for reception and translation of incoming messages, one for managing the conference data structures, and one for composing and broadcasting outgoing PDUs.

Conference Protocol Implementations. The conference protocol has been implemented on SUN SPARC workstations using a UNIX-like (SOLARIS) operating system, and it was programmed using the ANSI-C programming language. Furthermore, we used only standard UNIX inter-process and inter-machine communication facilities, such as uni-directional pipes and sockets.

A conference protocol implementation consists of the actual CPE which implements the protocol behaviour and a user-interface on top of it. We require that the user-interface is separated (loosely coupled) from the CPE to isolate the protocol entity; only the CPE is the object of testing. This is realistic because user interfaces are often implemented using dedicated software.

The conference protocol implementation has two interfaces: the CSAP and the USAP. The CSAP interface allows communication between the two UNIX processes, the user-interface and the CPE, and is implemented by two uni-

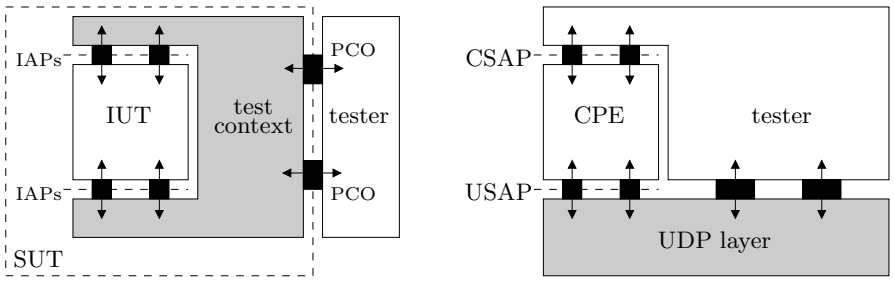
directional pipes. The USAP interface allows communication between the CPE and the underlaying layer UDP, and is implemented by sockets.

In order to guarantee that a conference protocol entity has knowledge about the potential conference partners the conference protocol entity reads a *configuration file* during the initialization phase.

Error seeding. For our experiment with automatic testing we developed 28 different conference protocol implementations. One of these implementations is correct (at least, to our knowledge), whereas in 27 of them a single error was injected deliberately. The erroneous implementations can be categorized in three different groups: *No outputs*, *No internal checks* and *No internal updates*. The group *No outputs* contains implementations that forget to send output when they are required to do so. The group *No internal checks* contains implementations that do not check whether the implementations are allowed to participate in the same conference according to the set of potential conference partners and the set of conference partners. The group *No internal updates* contains implementations that do not correctly administrate the set of conference partners.

4 TEST ARCHITECTURE

For testing a conference protocol entity (CPE) implementation, knowledge about the environment in which it is tested, i.e. the *test architecture*, is essential. A test architecture can (abstractly) be described in terms of a tester, an *Implementation Under Test* (IUT) (in our case the CPE), a test context, *Points of Control and Observation* (PCOs), and *Implementation Access Points* (IAPs) [8]. The test context is the environment in which the IUT is embedded and that is present during testing, but that is not the aim of conformance testing. The communication interfaces between the IUT and the test context are defined by IAPs, and the communication interfaces between the test context and the tester are defined by PCOs. The SUT (*System Under Test*) consists of the IUT embedded in its test context. Figure 1.3(a) depicts an abstract test architecture.



(a) Abstract test architecture (b) Test architecture for conference protocol entities

Figure 1.3 Test architecture

Ideally, the tester accesses the CPE directly at its IAPs, both at the CSAP and the USAP level. In our test architecture, which is the same as in [11], this is not the case. The tester communicates with the CPE at the USAP via the underlying UDP layer; this UDP layer acts as the test context. Since UDP behaves as an unreliable channel, this complicates the testing process. To avoid this complication we make the assumption that communication via UDP is reliable and that messages are delivered in sequence. This assumption is realistic if we require that the tester and the CPE reside on the same host machine, so that messages exchanged via UDP do not have to travel through the protocol layers below IP but 'bounce back' at IP.

With respect to the IAP at the CSAP interface we already assumed in the previous section that the user interface can be separated from the core CPE. Since the CSAP interface is implemented by means of pipes the tester therefore has to access the CSAP interface via the pipe mechanism.

Figure 1.3(b) depicts the concrete test architecture. The SUT consists of the CPE together with the reliable UDP service provider. The tester accesses the IAPs at the CSAP level directly, and the IAPs at USAP level via the UDP layer.

Formal model of the test architecture For formal test derivation, a realistic model of the behavioural properties of the complete SUT is required, i.e. the CPE and the test context, as well as the communication interfaces (IAPs and PCOs). The formal model of the CPE is based on the formal protocol specifications in Section 3. Using our assumption that the tester and the CPE reside on the same host, the test context (i.e. the UDP layer) acts as a reliable channel that provides in-sequence delivery. This can be modelled by two unbounded first-in/first-out (FIFO) queues, one for message transfer from tester to CPE, and one vice versa. The CSAP interface is implemented by means of pipes, which essentially behave like bounded first-in/first-out (FIFO) buffers. Under the assumption that a pipe is never 'overloaded', this can also be modelled as an unbounded FIFO queue. The USAP interface is implemented by means of sockets. Sockets can also be modelled, just as pipes, by unbounded FIFO queues. Finally, the number of communicating peer entities of the CPE, i.e. the set of potential conference partners, has been fixed in the test architecture to two. Figure 1.4 visualizes the complete formal model of the SUT.

To compare test derivation and test execution based on LOTOS, SDL and PROMELA, we have built models of the SUT in these three languages. In the LOTOS description of the SUT the behaviour description of the CPE is extended with queues that model the underlying UDP service, the pipes and the sockets. As two consecutive unbounded queues behave exactly equivalent as a single unbounded queue, an optimization was made with respect to such consecutive queues. A consequence of the introduction of queues is that the hidden synchronizations between the queues and the CPE at CSAP and USAP lead to internal steps. In Section 5 we will see the impact that these internal steps have on run-time and memory consumption of the on-the-fly tester.

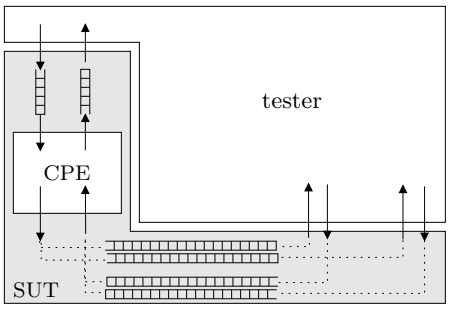


Figure 1.4 Formal model of the SUT

Similar to LOTOS, a complete specification of the SUT, including the underlying UDP layer and the interfaces, has been constructed in SDL. For automatic derivation of TTCN the use of certain SDL constructs had to be restricted, e.g. the built-in type PId for address construction had to be converted.

In PROMELA, also a model of the SUT has been constructed. In contrast to the LOTOS and SDL specifications, an optimization has been made in PROMELA that allows removal of the queues without changing the observable behaviour of the protocol.

5 TESTING ACTIVITIES

This section describes our testing activities. After summarizing the overall results we will elaborate on the test activities for each of the specification languages. We used the LOTOS and PROMELA specifications for on-the-fly test derivation and execution, using the correctness criterion **ioco** [12]. We used the SDL specification to derive and execute TTCN in batch mode.

For each of the specification languages we started with initial experiments to identify errors in the specifications and to test the (specification language specific) en/decoding functions of the tester. Once we had sufficient confidence in the specification and test tools, we tested the (assumed to be) correct implementation, after which the 27 erroneous mutants were tested by people who did not know which errors had been introduced in these mutants.

In the following table we summarize the results of these experiments. Each implementation was tested several times: on-the-fly with different seeds, and batch-wise with 13 different test cases. For each implementation and each FDT we indicate the verdict, and the minimal and maximal number of execution steps, i.e. LOTOS, PROMELA or TTCN test events that were taken to reach the verdict. If all tests led to a **pass** verdict, we indicate **pass**. For tests that have a **pass** verdict we do not indicate the number of steps. If at least one test led to a **fail** verdict we indicate **fail**. If no test led to a **fail** verdict, but at least one test led to an **inconclusive** verdict because of a timeout, we indicate

mutant nr.	*LOTOS* *verdict*	steps min	steps max	*Promela* *verdict*	steps min	steps max	*SDL* *verdict*	steps min
				'correct' implementation				
0	**pass**	-	-	**pass**	-	-	**pass**	-
				Incorrect Implementations – No outputs				
1	**fail**	37	66	**fail**	9	51	**pass**	-
2	**fail**	21	37	**fail**	6	116	**timeout**	7
3	**fail**	63	78	**fail**	24	498	**timeout**	7
4	**fail**	65	68	**fail**	20	83	**timeout**	7
5	**fail**	11	17	**fail**	2	10	**timeout**	7
6	**fail**	31	192	**fail**	14	81	**timeout**	7
				Incorrect Implementations – No internal checks				
7	**fail**	57	126	**fail**	31	392	**timeout**	12
8	**fail**	31	37	**fail**	38	200	**pass**	-
9	**pass**	-	-	**pass**	-	-	**timeout**	12
10	**pass**	-	-	**pass**	-	-	**pass**	-
				Incorrect Implementations – No internal updates				
11	**fail**	26	126	**fail**	29	143	**timeout**	12
12	**fail**	21	44	**fail**	6	127	**timeout**	7
13	**fail**	21	45	**fail**	6	19	**timeout**	7
14	**fail**	57	76	**fail**	28	146	**fail**	7
15	**fail**	207	304	**fail**	19	142	**fail**	17
16	**fail**	40	208	**fail**	25	83	**fail**	25
17	**fail**	35	198	**fail**	9	46	**timeout**	8
18	**fail**	31	238	**fail**	12	121	**timeout**	7
19	**fail**	29	467	**fail**	9	165	**pass**	-
20	**fail**	57	166	**fail**	33	142	**timeout**	7
21	**fail**	63	178	**fail**	15	219	**fail**	7
22	**fail**	57	166	**fail**	31	144	**timeout**	7
23	**fail**	21	35	**fail**	5	33	**fail**	7
24	**fail**	69	126	**fail**	31	127	**pass**	-
25	**fail**	37	55	**fail**	7	51	**timeout**	7
26	**fail**	66	91	**fail**	24	235	**pass**	-
27	**fail**	46	210	**fail**	23	139	**fail**	17

timeout (this only applies to the SDL-based tests). A **timeout** implies a deadlock, and therefore it must be viewed as a serious warning.

LOTOS To instantiate our test architecture with LOTOS specific components, we only needed to produce a specification-dependent EXPLORER component and specific en/decoding routines for the ADAPTER component; the remaining components of our test tool architecture could be reused. The EXPLORER module was automatically generated using the CADP tool set, see

section 2. The ADAPTER en/decoding routines were hand-written. The resulting tester is parameterized with the seed of its random number generator and the target depth.

We started by repeatedly running the tester in automatic mode, each time with a different seed for the random number generator, until either a depth of 500 steps was reached or an inconsistency between tester and implementation was detected (i.e. **fail**, usually after some 30 to 70 steps). This uncovered some errors in both the implementation and the specification, which were repaired. In addition we have run the tester in user-guided, manual mode to explore specific scenarios and to study failures that were found in fully automatic mode.

Once we had sufficient confidence in the quality of the specification and implementation we repeated the previous 'automatic mode' experiment, but now with a (target) depth of $1,000,000$ steps. This depth has not been reached, because the LOTOS EXPLORER-PRIMER module runs out of memory (without discovering an inconsistency) after a few thousand steps – the longest trace consisted of $27,803$ steps and took 1.4 Gb of memory. For this trace the computation of a single step took from 3 seconds (99.9% of the steps) up to 43 minutes of CPU time on a 296 MHz Sun UltraSPARC-II processor. The main reason for the huge memory consumption is that there are (known) memory leaks in the EXPLORER, while we could not use the garbage collector supplied in CADP because it does not cooperate with our PRIMER (this is being fixed). The main reason for the long computation time lies in the internal steps in the model of the CPE extended with test context (see Section 4), because our PRIMER has to explore large numbers of states for traces that contain long sequences of internal steps.

To test the error-detection capabilities of our tester we repeatedly ran the tester in automatic mode for a depth of 500 steps, each time with a different seed for the random number generator, on the 27 mutants. The tester was able to detect 25 of them. The two mutants that could not be detected accept PDUs from any source – they do not check whether an incoming PDU comes from a potential conference parter. This is not explicitly modeled in our LOTOS specification, and therefore these mutants are **ioco**-correct with respect to the LOTOS specification, which is why we can not detect them.

Promela To instantiate our tester with PROMELA-specific components, we generated a single module, which implements both the EXPLORER and PRIMER modules, automatically from the PROMELA specification using the TROJKA tool [13]. TROJKA is based on the SPIN tool for PROMELA [7]. We could reuse most of the ADAPTER developed for LOTOS; changes were only necessary in the parts that depend on the specification labels.

With the PROMELA based tester we repeated the experiments that we did with the LOTOS based one. We received the same results, but in a much shorter time (on average about 1.1 steps per second), and we were able to reach greater depths (450,000 steps), using less memory (400Mb). The differences with the LOTOS based tester can be explained by the use of memory-efficient

internal data representations and the use of hashing techniques to remember the results of unfoldings. These techniques were inherited by TROJKA from SPIN. The PROMELA based tester was able to detect the same 25 of the 27 mutants as the LOTOS based tester (which is no surprise because both testers check for the same correctness criterion **ioco**).

SDL We used the TAU tool kit for batch derivation and execution of test suites in TTCN from SDL [1]. The functionality of the EXPLORER, PRIMER and (batch derivation) DRIVER were covered by TAU's AUTOLINK test derivation tool, which generates TTCN test suites from the SDL specification, guided by MSCs that were derived manually from the SDL specification using TAU's SDL simulator.

Before running the derived TTCN test suite against our implementations, we ran it against the original SDL specification to validate the TTCN test suite, by running the SDL simulator and TAU's TTCN simulator in connection. This uncovered some problems in AUTOLINK. Another problem encountered was that for some MSCs no TTCN suites could be derived, although the MSC could be successfully verified (this depends on the memory of our computers). An interesting aspect is that the partial temporal ordering of the events in the MSC is sometimes not respected in the TTCN code derived by AUTOLINK. Finally, we have experienced some minor problems and bugs. Some of the abovementioned problems may have been solved already in recent releases of the software involved.

The batch execution DRIVER module was automatically generated from the complete TTCN suites using the TTCN compiler of TAU. This DRIVER is linked with an ADAPTER module using the GCI interface; the result is a single program that can execute the test suites (see Section 2). Some parts of the en/decoding functions of the ADAPTER could be generated using KIMWITU [3]; the rest consists of hand-written C code.

We have derived 15 test cases (MSCs) from the SDL specification. The time to build an MSC by means of simulation is 3 minutes for an MSC with 7 events which includes the time to split it into parts (two parts in this case). This was the shortest time. The longest time was 45 minutes for 44 events (15 parts). The average time was 12 minutes for an average of 20 events and 5 parts. The test derivation time consists of the abovementioned times plus the AUTOLINK generation step's time which was less than 8 seconds. The preceding splitting turned out to be essential for most cases (e.g. 14 minutes without splitting becomes 7 seconds *with* splitting). For two of these MSCs no TTCN could not be derived, not even after splitting, because AUTOLINK ran out of memory.

We will try to sketch our informal strategy for defining test purposes in the next few lines. Most of the test purposes are concerned with a single conference. Various arbitrary interleavings of join actions, data transfer and leave actions give rise to one test purpose each. The other test purposes check the absence of interference between two simultaneous conferences (for 3 users it makes no sense to have more than 2 conferences).

The test execution time, running the TTCN which was derived in a batch-wise way against the implementation, took from 2 to 5 seconds.

The detection of errors was done by repeating all the 13 test cases for which TTCN could be derived for the 27 mutants. Six **fail** verdicts were obtained, next to 15 **inconclusive** verdicts that were the effect of a timeout. We felt that such **inconclusive** verdicts ought to be viewed a serious warning, because they indicate a deadlock. Six errors went undetected, although some of them could have been found by a larger test suite.

6 EVALUATION AND CONCLUSIONS

Conclusions. In this paper we have studied the feasibility of automatic test derivation and execution from a number of formal specifications. To conduct this study, a protocol has been modelled in three formal specification languages: LOTOS, PROMELA and SDL. Also, a set of concrete implementations has been constructed, some of which were injected with faults that were unknown to the person that performed the testing. To test these implementations based on the formal specifications an open test architecture has been defined that was successfully instantiated for on-the-fly testing with LOTOS and PROMELA and batch-wise testing with SDL. The results have been compared with respect to the number of erroneous implementations that could be detected for each of the specifications, and the time and effort that it took to test the implementations.

The tool architecture that was used to conduct the experiment supports both on-the-fly testing and batch testing. Several existing tool sets, such as CADP and TAU, were used as plug-ins for the tool architecture, thereby illustrating its openness.

In the on-the-fly approach, tests were fully automatically derived (in a random way) and executed. In the batch approach the construction of tests needed manual assistance. Execution in the batch approach was done automatically. Both the on-the-fly approach and the batch approach were able to detect many erroneous implementations. Using the on-the-fly techniques all erroneous implementations could be detected, except for those that contained errors that simply could not be detected due to modelling choices in the specification and the choice of implementation relation (hence, were formally no errors). Using batch testing based on SDL fewer erroneous implementations were detected. On the one hand this is caused by the occurrence of timeouts (which should be considered as indications of potential errors such as deadlocks), and on the other hand by the fact that less tests were executed due to the fact that manual assistance during test derivation was needed. By deriving more test cases in the batch approach it will be possible to increase the error detecting capability. Although with batch SDL testing certainly less erroneous implementations were detected, the current experiments are too restricted to deduce general conclusions from them. More experiments are needed where also more care is taken that the starting points are comparible.

In the on-the-fly approach, tests were derived using random selection of inputs. How many steps it takes to trigger errors depends very much on the

random choices that the tester makes, and the seed of the random choice generator. However, the results in this paper support the assumption that if the tester runs 'sufficiently long' then eventually all errors will be found. In the batch approach more human assistance is needed. Consequently, an error can often be found in less steps using the batch approach than in the on-the-fly approach. We found that for long traces that lead to errors, it is difficult to analyse the exact conditions under which such an error was triggered. Especially in the on-the-fly approach, error analysis was more difficult because no support was available to diagnose the trace that led to the error. Analysis could be made easier if the traces that lead to an error could be transformed into a form in which they can be studied in the existing development environments for the FDTs.

Further work. The experimental results that are presented in this paper are based on a case study of a single protocol and a limited number of implementations. To obtain more valuable results the number of cases studies and the number of experiments per case study should be increased. To enable a rigorous comparison of test derivation and test execution tools by different vendors, one (or more) case studies containing specifications and sets of implementations should be made publically available so that they can be used by several tool vendors to compare their test derivation/execution tools. The case study in this paper can be seen as one of the first initiatives towards such a *test tool benchmarking* activity. To promote this kind of benchmarking, the description of our case study together with the formal specifications and all implementations are available on the Web [9]. Everybody is thus invited to conduct and publish analogous experiments with his or her favourite test tools.

The on-the-fly approach for test derivation is currently implemented using a random strategy. Therefore, it is difficult to steer the test that is being derived and executed. In practice, more advanced and user controlled strategies that allow for the derivation of tests that are targeted towards specific test purposes, or that avoid deriving the same tests more than once, are needed. This requires research in the field of test purpose oriented test derivation, and adaptive test derivation techniques. The batch approach is currently limited by the fact that human intervention is needed to derive tests, and due to the fact that sometimes inconclusive verdicts are reached as a result of timeouts. More advanced batch test derivation techniques might overcome these deficiencies.

Most of the work to instantiate the on-the-fly tester goes into the making and validation of the formal specification. After this the specification module can be generated fully automatically. For the batch approach the most laborious task is to derive tests. Also, a laborious task is to produce the ADAPTER. Creation of the ADAPTER can be eased by developing a general framework in which the method to connect to the IUT is orthogonal to the data en/decoding, and to allow easy reuse of connection modules for common protocols. In addition, it is worth studying whether it is possible to develop languages and tools to specify the data en/decoding functions in a compact syntax and generate automatically code from that.

References

[1] Telelogic AB. Telelogic TAU Documentation, 1998.

[2] F. Brady and R.M. Barker. Infrastructural Tools for Information Technology and Telecommunications Conformance Testing, INTOOL/GCI, Generic Compiler/Interpreter (GCI) Interface Specification, Version 2.2, 1996. INTOOL doc. nr. GCI/NPL038v2.

[3] P. van Eijk, A. Belinfante, H. Eertink, and H. Alblas. The Term Processor KIMWITU. In E. Brinksma, editor, *TACAS'97*, pages 96–111. LNCS 1217, Springer-Verlag, 1997.

[4] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using On-the-Fly Verification Techniques for the generation of test suites. In R. Alur et al., editor, *CAV'96*. LNCS 1102, Springer-Verlag, 1996.

[5] L. Ferreira Pires. Protocol Implementation: Manual for Practical Exercises 1995/1996. Lecture notes, University of Twente, The Netherlands, 1995.

[6] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *TACAS'98*, pages 68–84. LNCS 1384, Springer-Verlag, 1998.

[7] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall Inc., 1991.

[8] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Framework: Formal Methods in Conformance Testing*. CD 13245-1, ITU-T Z.500. ISO – ITU-T, Geneve, 1996.

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

[10] M. Schmitt, A. Ek, B. Koch, J. Grabowski, and D. Hogrefe. – AUTOLINK – Putting SDL-based Test Generation into Practice. In A. Petrenko et al., editor, *IWTCS'98*, pages 227–243. Kluwer Academic Publishers, 1998.

[11] R. Terpstra, L. Ferreira Pires, L. Heerink, and J. Tretmans. Testing theory in practice: A simple experiment. In T. Kapus et al., editor, *COST 247 Workshop on Applied Formal Methods in System Design*, pages 168–183. University of Maribor, Slovenia, 1996.

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

[13] R.G. de Vries and J. Tretmans. On-the-Fly Conformance Testing using SPIN. In G. Holzmann et al., editor, *Fourth SPIN Workshop*, ENST 98 S 002, pages 115–128. Ecole Nationale Supérieure des Télécommunications, Paris, France, November 2 1998.