Automatic Testing with Formal Methods

Jan Tretmans and Axel Belinfante

University of Twente *
Department of Computer Science
Formal Methods and Tools group
P.O. Box 217, 7500 AE Enschede
The Netherlands

{tretmans,belinfan}@cs.utwente.nl

Abstract

The use of formal system specifications makes it possible to automate the derivation of test cases from specifications. This allows to automate the whole testing process, not only the test execution part of it. This paper presents the state of the art and future perspectives in testing based on formal methods. The theory of formal testing is briefly outlined, a test tool is presented which automates both test derivation and test execution *on-the-fly*, and an application case study is discussed.

1 Introduction

Testing is an important activity for checking the correctness of system implementations. It is performed by applying test experiments to an implementation under test, by making observations during the execution of the tests, and by subsequently assigning a verdict about the correct functioning of the implementation. The correctness criterion that is to be tested should be given in the system specification. The specification prescribes what the system has to do and what not, and, consequently, constitutes the basis for any testing activity.

During testing many problems may be encountered. These problems can be organizational, i.e., concerning the management of the test process, technical, i.e., concerning insufficient techniques and support for test specification or test execution leading to a laborious, manual and error-prone testing process, or financial, i.e., testing takes too much time and too many efforts – in some software developments projects testing may consume up to 50% of the project resources. Dealing with these testing problems is not always simple.

While analysing these issues it can be observed that many of the problems attributed to testing are actually problems of system specification. Many problems in the testing process occur because specifications are unclear, imprecise, incomplete and ambiguous. Without a specification which clearly, precisely, completely and unambiguously prescribes how a system implementation shall behave, any testing will be very difficult because it is unclear what to test. A bad system specification as starting point for the testing process usually leads to problems such as difficulties of

*This work 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.

interpretation and required clarifications of the specification's intentions. This leads to reworking of the specification during the testing phase of software development.

But also with a clear and precise specification the testing process may take too much effort, both in terms of time and money. Automation of testing activities then seems a logical solution. Automation may help in making the testing process faster, in making it less susceptible to human error by automating routine or error-prone tasks, and in making it more reproducible by making it less dependent on human interpretation.

There are many test tools available nowadays. Most of these test tools support the test execution process. This includes the execution, systematic storage and re-execution of specified test cases. Test cases have to be written down – manually – in a special language for test scripts which is usually tool specific. These test scripts can then be executed automatically. An alternative approach is *capture & replay*: while the tests are executed manually, they are recorded so that later they can be replayed several times. The advantages of automation of test execution are mainly achieved when tests have to be re-executed several times, e.g., during regression testing.

Test execution tools do not help in developing the test cases. Test cases have to be developed by clever humans, who, while reading and studying specifications, think hard about what to test and about how to write test scripts that test what they want to test. There are not many tools available that can help with, let alone automate, the generation of good tests from specifications. Yet, also test generation, i.e., the activity of systematically and efficiently developing tests from specifications, is a laborious, manual, and error-prone process. One of the main bottlenecks for automating the test generation process is the shape and status of specifications. In the first place, many current-day specifications are unclear, incomplete, imprecise and ambiguous, as explained above, which is not a good starting point for systematic development of test cases. In the second place, current-day specifications are written in natural language, e.g., English, German, etc. Natural language specifications are not easily amenable to tools for automatic derivation of the test cases.

Formal methods Formal methods are concerned with mathematical modelling of software and hardware systems. Due to their mathematical underpinning formal methods allow to specify systems with more precision, more consistency and less ambiguity. Moreover, formal methods allow to formally simulate, validate and reason about system models, i.e., to prove with mathematical precision the presence or absence of particular properties in a design or specification. This makes it possible to detect deficiences earlier in the development process. An important aspect is that specifications expressed in a formal language are much easier processable by tools, hence allowing more automation in the software development trajectory.

Formal methods are more and more used in software engineering, see e.g., [HB95]. In [CTW99] we reported about the use of formal methods in the Bos system which was developed by CMG Den Haag B.V. In the Bos project the formal methods Z and Promela were used for specification of the design. Promela is a formal language for modelling communication protocols, which is based on automata theory [Hol91]. Z is a formal language based on set theory and predicate logic [Spi92].

In [GWT98] we reported about the benefits which were obtained in the testing phase of the Bos project by the use of formal methods. One of the main conclusions of that contribution was that using formal methods in the software development trajectory is very beneficial in the testing phase. These benefits are due to the clarity, preciseness, consistency and completeness of formal specifications, which make that test cases can be efficiently, effectively and systematically derived from the formal specifications. Even if all test generation in the Bos project was manual and not supported by tools, an improvement in quality and costs of testing was achieved. It was noted in [GWT98] that further improvements seem possible by automation of the test generation process, but that more research and development would be necessary to make such automatic derivation of test cases from formal specifications feasible.

Goal The goal of this paper is to explore some developments and the state of the art in the area of automatic derivation of test cases from formal specifications. In particular, this contribution presents a glimpse of the sound, underlying formal testing theory, the test derivation tool TorX implementing this theory, and an application of automatic test derivation using TorX. Moreover, we discuss how the use of formal methods may improve the testing process, and how the testing process can help in introducing formal methods in software development.

The test tool TorX is a prototype tool which integrates automatic test derivation and test execution. TorX is developed within the project *Côte de Resyste*, which is a joint project of the University of Twente, Eindhoven University of Technology and Philips Research Laboratories Eindhoven, and which is supported by the Dutch Technology Foundation STW. The goal of *Côte de Resyste* is to supply methods and tools for the complete automation of conformance testing of reactive system implementations based on formal specifications. This paper presents the state of the art, sketches the perspectives and discusses some of the hurdles on the road to completely automatic testing.

2 Testing based on Formal Methods

Testing Testing is an operational way to check the correctness of a system implementation by means of experimenting with it. Tests are applied to the implementation under test in a controlled environment, and, based on observations made during the execution of the tests, a verdict about the correct functioning of the implementation is given. The correctness criterion that is to be tested is given by the system specification; the specification is the basis for testing.

Conformance testing There are many different kinds of testing. In the first place, different aspects of system behaviour can be tested: Does the system have the intended functionality and does it comply with its functional specification (functional tests or conformance tests)? Does the system work as fast as required (performance tests)? How does the system react if its environment shows unexpected or strange behaviour (robustness tests)? Can the system cope with heavy loads (stress testing)? How long can we rely on the correct functioning of the system (reliability tests)? What is the availability of the system (availability tests)?

Moreover, testing can be applied at different levels of abstraction and for different (sub-)systems: individual functions, modules, combinations of modules, subsystems and complete systems can all be tested. Another distinction can be made according to the parties or persons performing (or responsible for) testing. In this dimension there are, for example, system developer tests, factory acceptance tests, user acceptance tests, operational acceptance tests, and third party (independent) tests, e.g., for certification.

A very common distinction is the one between black box and white box testing. In black box testing, or functional testing, only the outside of the system under test is known to the tester. In white box testing, also the internal structure of the system is known and this knowledge can be used by the tester. Naturally, the distinction between black and white box testing leads to many gradations of grey box testing, e.g., when the module structure of a system is known, but not the code of each module.

In this paper, we concentrate on black box, functional testing. We do not care about the level of (sub-)systems or who is performing the testing. Key points are that there is a system implementation exhibiting behaviour and that there is a specification. The specification is a prescription of what the system should do; the goal of testing is to check, by means of testing, whether the implemented system indeed satisfies this prescription. We call this kind of testing *conformance testing*.

Conformance testing with formal methods When using formal methods in conformance testing we mean that we check conformance, by means of testing, of a black-box implementation with respect to a formal specification, i.e., a specification given in a formal specification language.

We will concentrate on formal specification languages that are intended to specify *reactive systems*, i.e., software systems for which their main behaviour consists of reacting with responses to stimuli from their environment. Concurrency and distribution usually play an important rôle in such systems. Examples of reactive systems are communication protocols and services, embedded software systems and process control systems.

In this paper, we concentrate on the formal language PROMELA as our specification language [Hol91]. PROMELA is a formal language for modelling communication protocols, it is based on automata theory [Hol91], and it is the input language for the model checker SPIN [Spi]. PROMELA was used as one of the specification languages in the BOS project [GWT98].

Other methods and tools for formal conformance testing have been developed, e.g., for Abstract Data Type specifications [Gau95], for Finite State Machines [LY96], for SDL [SEK+98] and for LOTOS [Bri88, BFV+99].

Testing and verification Formal testing and formal verification are complementary techniques for analysis and checking of correctness of systems. While formal verification aims at proving properties about systems by formal manipulation on a mathematical model of the system, testing is performed by exercising the real, executing implementation (or an executable simulation model). Verification can give certainty about satisfaction of a required property, but this certainty only applies to the model of the system: any verification is only as good as the validity of the system model. Testing, being based on observing only a small subset of all possible instances of system behaviour, can never be complete: testing can only show the presence of errors, not their absence. But since testing can be applied to the real implementation, it is useful in those cases when a valid and reliable model of the system is difficult to build due to complexity, when the complete system is a combination of formal parts and parts which cannot be formally modelled (e.g., physical devices), when the model is proprietary (e.g., third party testing), or when the validity of a constructed model is to be checked with respect to the physical implementation.

The conformance testing process In the process of conformance testing there are two main phases: *test generation* and *test execution*. Test generation involves analysis of the specification and determination of which functionalities will be tested, determining how these can be tested, and developing and specifying test scripts. Test execution involves the development of a test environment in which the test scripts can be executed, the actual execution of the test scripts and analysis of the execution results and the assignment of a verdict about the well-functioning of the implementation under test.

The formal conformance testing process Also in conformance testing based on formal methods a test generation phase and a test execution phase can be distinguished. The difference is that now a formal specification is the starting point for the generation of test cases. This allows to automate the generation phase: test cases can be derived algorithmically from a formal specification following a well-defined and precisely specified algorithm. For well-definedness of the algorithm it is necessary that it is precisely defined what (formal) conformance is. Well-defined test derivation algorithms guarantee that tests are *valid*, i.e., that tests really test what they should test. Section 3 presents a glimpse on the theoretical background of formal conformance testing and algorithmic test derivation.

In principle, test execution does not depend on how the tests are generated. Existing test execution tools can be used. Analysis of results and verdict assignment are easily automated using a formal

specification. Moreover, automatic derivation of tests allows to combine test derivation and test execution: tests can be executed while they are derived. We call this way of interleaved test derivation and test execution *on-the-fly* testing. It is the way that the test tool TorX works; it will be further discussed in section 4.

3 A Glimpse of Formal Testing Theory

There are different theories of formal testing. In this section we concentrate on those theories which are important for Promela-based testing, in particular, the so-called **ioco** testing theory for labelled transition systems. The intention of this section is to give an impression of this theory of formal testing and to show that there is a well-defined and sound formal underpinning for the test derivation algorithms. This section can be easily skipped by those who are not interested in the underpinning of formal testing, while those who are really interested in the details should not rely on the presentation given in this section, but should consult the literature for full details, algorithms and proofs [Tre96, Hee98, VT98, Tre99].

Labelled transition systems Labelled transition systems provide a formalism to give a semantics to Promela specifications. A labelled transition system consists of states and labelled transitions between states. The states model the system states; the labelled transitions model occurrences of interactions of the system with its environment, e.g., input or output. We write $s \xrightarrow{a} s'$ if there is a transition labelled a from state s to state s'. This is interpreted as: "when the system is in state s it may perform interaction a and go to state s' ". Interactions can be concatenated using the following notation: $s \xRightarrow{a \cdot b \cdot c} s''$ expresses that the system, when in state s may perform the sequence of actions abc and end in state s''.

Figure 1 shows two labelled transition systems r_1 and r_2 modelling candy machines, with actions in $\{?but, !choc, !liq\}$. For example, r_1 may produce chocolate *!choc* after pushing the button *?but* twice:

$$r_1 \xRightarrow{?but \cdot ?but \cdot !choc}$$

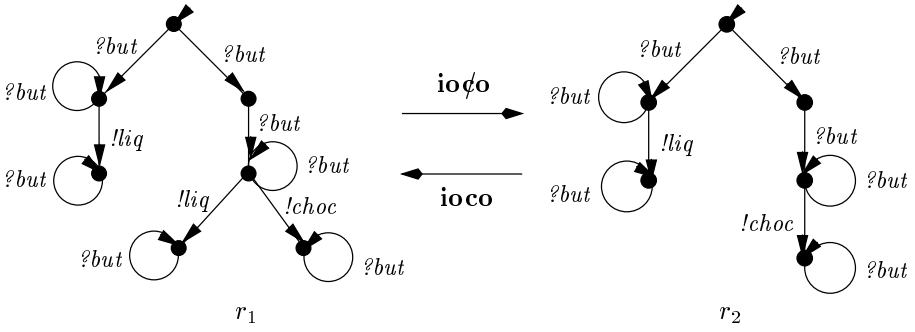


Figure 1: Labelled transition systems

Input-output transition systems A special class of labelled transition systems is formed by *input-output transition systems*. We assume that actions can be partitioned into input actions L_I and output actions L_U. Moreover, we require that input actions are always enabled. In terms of

transition systems: for all states s and for all input action $?a \in L_I$: $s \xrightarrow{?a} s'$ for some state s'. In input-output transition systems, inputs of one system communicate with the outputs of the other system, and vice versa. In particular, the inputs of a system are the outputs of a tester testing that system. We denote input actions with $?a$ and output actions with $!x$.

The example transition systems in figure 1 are both input-output transition systems when $L_I = \{?but\}$ and $L_U = \{!choc, !liq\}$: for all states s of r_1 and r_2 we can always find some s' such that $s \xrightarrow{?but} s'$.

Conformance The major issue of conformance testing is to decide whether an implementation is correct with respect to a specification. This requires a notion of conformance, which is covered by defining an *implementation relation*. An implementation relation is a relation between the domain of specifications and the domain of models of implementations, such that (i, s) is in the relation if and only if implementation i is a conforming implementation of specification s.

Our specifications are expressed in PROMELA which we semantically interpret as labelled transition systems. As implementations we consider input-output transition systems. Now we express conformance by defining the implementation relation **ioco** between input-output transition systems and labelled transition systems. Let i be an input-output transition system (the implementation) and let s be a labelled transition system (the (semantic model of the) specification), then

$$i \textbf{ ioco } s \quad \Longleftrightarrow_{\text{def}} \quad \forall \sigma \in \mathit{Straces}(s) : \quad \mathit{out}(\, i \textbf{ after } \sigma \,) \subseteq \mathit{out}(\, s \textbf{ after } \sigma \,)$$

where

- $p \textbf{ after } \sigma \quad =_{\text{def}} \quad \{p' \mid p \stackrel{\sigma}{\Longrightarrow} p'\}$
  $p \textbf{ after } \sigma$ is the set of states in which transition system p can be after having executed the sequence of actions σ;
- $\mathit{out}(p) \quad =_{\text{def}} \quad \{\, x \in L_U \mid p \xrightarrow{x} \,\} \cup \{\delta \mid \forall x \in L_U : p \not\xrightarrow{x} \}$
  $\mathit{out}(p)$ is the set of possible output actions in state p or it is $\{\delta\}$ if no output action is possible in state p; δ is a special action modelling *quiescence*, i.e., the absence of outputs; it is usually implemented as a time-out;
- $\mathit{out}(\, p \textbf{ after } \sigma \,) \quad =_{\text{def}} \quad \bigcup \{\, \mathit{out}(p') \mid p' \in p \textbf{ after } \sigma \,\}$
  $\mathit{out}(\, p \textbf{ after } \sigma \,)$ is the set of output actions which may occur in some state of $p \textbf{ after } \sigma$;
- $\mathit{Straces}(s) \quad =_{\text{def}} \quad \{\sigma \in (L_I \cup L_U \cup \{\delta\})^* \mid s \stackrel{\sigma}{\Longrightarrow}\}$
  $\mathit{Straces}(s)$ is the set of suspension traces of the specification s, i.e., the sequences of input actions, output actions and quiescence which s may execute.

Informally, an implementation i is **ioco**-correct with respect to the specification s if i can never produce an output which could not have been produced by s in the same situation, i.e., after the same sequence of actions (suspension trace). Moreover, i may only be quiescent, i.e., produce no output at all, if s can do so.

In figure 1 we have that r_2 **ioco** r_1, but not r_1 **ioco** r_2; r_1 is not a correct implementation of specification r_2 since r_1 may produce chocolate *!choc* after the trace $?but \cdot \delta \cdot ?but$ while r_2 can't. Formally: $!choc \in \mathit{out}(\, r_1 \textbf{ after } ?but \cdot \delta \cdot ?but \,)$ and $!choc \notin \mathit{out}(\, r_2 \textbf{ after } ?but \cdot \delta \cdot ?but \,)$.

More formal definitions, proofs, explanations and a rationale for the use of **ioco** as implementation relation can be found in [Tre96]. This paragraph was only intended to give some idea about the style of definitions and the level of formality used. Note, however, that it is very important to define such an implementation relation, expressing the notion of conformance, in a formal way. Without such a definition it is impossible to formally reason about validity of test derivation algorithms.

Testing An algorithm for the derivation of tests from labelled transition systems or PROMELA specifications should be devised in such a way that all and only all **ioco**-erroneous implementations will have as the result the verdict **fail**.

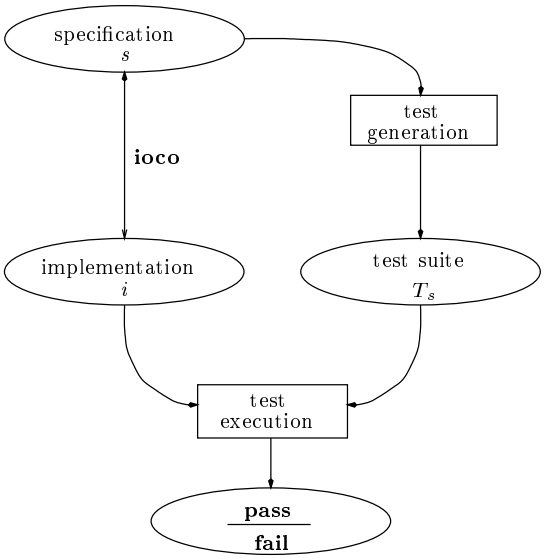


Figure 2: Conformance and test derivation

In figure 2, the formal testing process is expressed as follows. Starting with a formal specification s, an implementation i has been developed by some programmer; it is, for example, a C program. By means of testing we would like to check whether i is correct with respect to s, i.e., whether i **ioco** s. To this extent, a test suite T_s is generated from the same specification s following a test generation algorithm T. Subsequent execution of the test suite T_s with the implementation i, denoted by $\mathit{test\_exec}(T_s, i)$, leads to a verdict, either **pass** or **fail**. **pass** indicates that no evidence of non-conformance was found; **fail** indicates that an error was found. Now, if we want to draw a valid conclusion about conformance from the resulting verdict, there should be a relation between **ioco** and T, in the following sense:

$$\forall i, s : \ i \ \mathbf{ioco} \ s \iff \mathit{test\_exec}(T_s, i) = \mathbf{pass}$$

If this holds we can conclude from a successful test campaign, i.e., a campaign with verdict **pass**, that the implementation is correct, and moreover, only from a successful campaign this can be concluded.

Unfortunately, in practice we have to do with less: performing sufficiently many tests in order to be sure that an implementation is correct is not feasible, because this usually requires infinitely many tests. In this case we have a weaker requirement corresponding to the left-to-right implication of the above equation: if the result of test execution is **fail** then we are sure that the implementation is not **ioco**-correct. This requirement on test suites is called *soundness*. It is a minimal, formal requirement on test suites in order to be able to draw any useful conclusion from any testing campaign.

Test derivation We now present, in an informal way, a test derivation algorithm for **ioco**-test derivation taken from [Tre96]. The algorithm is recursive, i.e., it repeats itself, and it is nondeterministic, i.e., different choices in the algorithm can be made which lead to different valid test cases. The algorithm generates test cases which are labelled transition systems themselves, but

with a special structure: (*i*) a test case is a finite and tree-structured labelled transition system; and (*ii*) each terminal state of a test case is labelled either **pass** or **fail**; and (*iii*) in each non-terminal state of a test case either there is one transition labelled with a system input, or there are transitions for all possible system outputs and one special transition labelled θ.

Execution of a test case with an implementation under test corresponds to the simultaneous execution of transitions labelled with the same name. If test execution terminates in a test-case state labelled **fail** then the verdict **fail** is assigned.

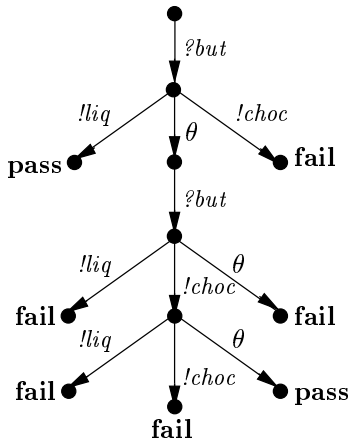


Figure 3: An example test case

Figure 3 gives an example of a test case, which specifies that as the first action the input *?but* must be supplied to the implementation under test. The special transitions labelled θ model a time-out: this transition will be taken if none of the output responses can be observed; the implementation under test is quiescent. Test execution of this test case with the system r_1 from figure 1 results in the verdict **fail**: the sequence of interactions $?but \cdot \theta \cdot ?but \cdot !liq$ leads to a terminal state of the test case labelled **fail**.

Test derivation algorithm Let s be a labelled transition system specification with initial state s_0. Let S be a non-empty set of states, with initially $S = \{s_0\}$. S represents the set of all possible states in which the implementation can be at the current stage of the test case.

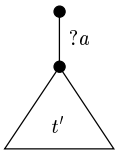
A test case t is obtained from S by a finite number of recursive applications of one of the following three nondeterministic choices:

1. t :=

   The single-state test case **pass** is always a valid test case. It terminates the recursion in the algorithm.

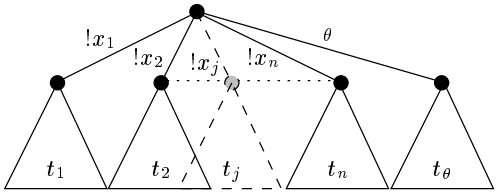
2. t :=

where $?a \in L_I$, S **after** $?a \neq \emptyset$, and t' is obtained by recursively applying the algorithm for $S' = S$ **after** $?a$.

Test case t supplies the stimulus $?a$ to the implementation under test and subsequently behaves as test case t'. t' is obtained by applying the algorithm recursively to S' which is the set of specification states which can be reached via an $?a$ transition from a state in S.

3. t :=



where $L_U = \{x_1, x_2, \ldots, x_n\}$, $1 \leq j \leq n$:

if	$x_j \notin out(S)$	then	$t_j =$ **fail**
if	$\delta \notin out(S)$	then	$t_\theta =$ **fail**
if	$x_j \in out(S)$	then	t_j is obtained by recursively applying the algorithm for S **after** x_j
if	$\delta \in out(S)$	then	t_θ is obtained by recursively applying the algorithm for $\{s \in S \mid s \xrightarrow{\delta}\}$.

Test case t checks the next output response of the implementation; if it is an unvalid response, i.e., $x \notin out(S)$ then the test case terminates in **fail**; if it is a valid response the test case continues recursively. The observation of quiescence δ is treated separately by the time-out action θ.

This algorithm was proved in [Tre96] to produce only *sound* test cases, i.e., test cases which never produce **fail** while testing an **ioco**-conforming implementation. Moreover, it was shown that any non-conforming implementation can always be detected by a test case generated with this algorithm, hence the algorithm is *exhaustive*. This algorithm is implemented for PROMELA specifications in the test tool TORX; this is described in section 4.

The reader is invited to check that the example test case of figure 3 is obtained by applying the test derivation algorithm to specification r_2 of figure 1. Test execution of the test case with r_1 will result in assigning the verdict **fail** as explained above. This is consistent with the fact that r_1 **ioco** r_2.

4 Tools

The formal test theory and the **ioco**-test derivation algorithm, of which a glimpse was presented in section 3, fortunately have a much wider and more practical applicability than the testing of candy machines. Different test tools have been built using this algorithm. These tools are able to automatically derive tests from formal system specifications. These include TVEDA [Pha94, Cla96], TGV [FJJV97] and TORX [BFV+99].

TVEDA is a tool, developed at France Telecom CNET, which is able to generate test cases in TTCN from formal specifications in the language SDL. The formal language SDL is an ITU-T standard and is often used for specification and design of telecommunication protocols [CCI92]; TTCN is an ITU-T and ISO standard for the notation of test suites [ISO91, part 3]. France Telecom uses TVEDA to generate tests for testing of telecom products, such as ATM protocols.

The tool TGV generates tests in TTCN from LOTOS or SDL specifications. LOTOS is a specification language for distributed systems standardized by ISO [ISO89]. TGV allows test purposes to be specified by means of automata, which makes it possible to identify the parts of a specification which are interesting from a testing point of view. The prototype tools TVEDA and TGV are currently integrated into the SDL tool kit OBJECTGEODE [KJG99].

Whereas TVEDA and TGV only support the test derivation process by deriving test suites and expressing them in TTCN, the tool TORX combines **ioco**-test derivation and test execution in an integrated manner. This approach, where test derivation and test execution occur simultaneously, is called *on-the-fly testing*. Instead of deriving a complete test case, the test derivation process only derives the next test event from the specification and this test event is immediately executed. While executing a test case, only the necessary part of the test case is considered: the test case is derived *lazily* (cf. lazy evaluation of functional languages; see also [VT98]). The principle is depicted in figure 4. Each time the *Tester* decides whether to trigger the *IUT* (Implementation Under Test) with a next stimulus or to observe the output produced by the *IUT*. This corresponds to the choice between 2. and 3. in the test derivation algorithm of section 3. If a stimulus is given to the *IUT* (choice 2.) the *Tester* looks into the system specification – the *specification module* – for a valid stimulus and offers this input to the *IUT* (after suitable translation and encoding). When the *Tester* observes an output or observes that no output is available from the *IUT* (called quiescence in the **ioco**-theory and usually observed as a time-out), it checks whether this response is valid according to the specification (choice 3.). This process of giving stimuli to the *IUT* and observing responses from the *IUT* can continue until an output is received which is not correct according the specification resulting in a **fail**-verdict. For a correct *IUT* the only limits are the capabilities of the computers on which TORX is running. Test cases of length up to 450,000 test events (stimuli and responses) have been executed completely automatically.

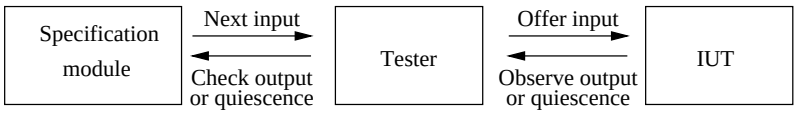


Figure 4: On-the-fly testing

TORX is currently able to derive test cases from LOTOS and PROMELA specifications, i.e., the specification module in figure 4 can be instantiated with a LOTOS or a PROMELA module. The LOTOS implementation is based on CÆSAR [Gar98]; the PROMELA implementation is based on the model checker SPIN [Hol91, Spi]. But since the interface between the specification module and the rest of the tool uses the OPEN/CÆSAR interface [Gar98] for traversing through a labelled transition system, the tool can be easily extended to any formalism with transition system semantics for which there is an OPEN/CÆSAR interface implementation available.

An important aspect in the communication between the *Tester* and the *IUT* is the encoding and decoding of test events. Test events in specifications are abstract objects which have to be translated into some concrete form of bits and bytes to communicate with the *IUT*, and vice versa. These en-/decoding functions currently have to be written manually, still; this is a laborious task, but, fortunately, it needs to be done only once for each *IUT*.

The *Tester* can operate in a manual or automatic mode. In the manual mode, the next test event – input or output and, if an input, the selection of the input – can be chosen interactively by the TORX user. In the automatic mode everything runs automatically and selections are made randomly. A seed for random number generation can then be supplied as a parameter. Moreover, a maximum number for the length of the test cases may be supplied. Section 5 will elaborate on a particular example system tested with TORX, thus illustrating the concepts presented here.

5 Conference Protocol Example

In the context of the *Côte de Resyste* project we have done a case study to test implementations of a simple chatting protocol, the Conference Protocol [FP95, TFPHT96]. In this case study an implementation of the conference protocol has been built (in C, based on the informal description of the protocol), from which 27 mutants have been derived by introducing single errors. All these 28 implementations have been tested using TorX, by persons who did not know which errors had been introduced to make the mutants. This case study is described in greater detail in [BFV+99].

The remainder of this section is structured as follows. Section 5.1 gives an overview of the conference protocol and the implementations we made. Section 5.2 discusses the test architecture that we used for our testing activities, which are described in Section 5.3.

Availability on WWW An elaborate description of the Conference Protocol, together with the complete formal specifications and our set of implementations can be found on the web [CdR]. We hereby heartily invite you to repeat our experiment with your favorite testing tool!

5.1 The Conference Protocol

Informal description The conference service provides a multicast service, resembling a 'chat-box', 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 the 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 at USAP (UDP Service Access Point) to provide the conference service at CSAP (Conference Service Access Point). 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.

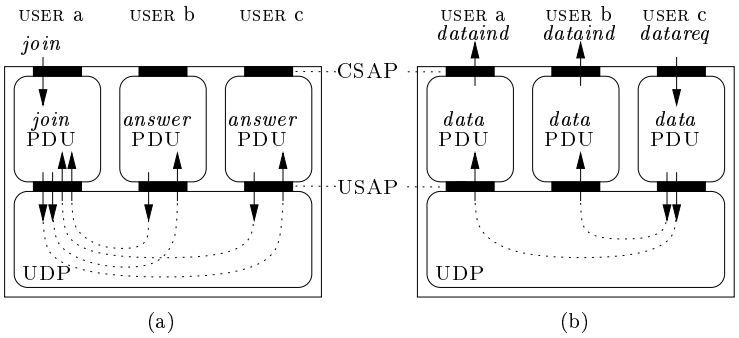


Figure 5: The conference protocol

Figure 5 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 specification in PROMELA The protocol has been specified in PROMELA. Instantiating this specification with three potential conference users, a PROMELA model for testing is generated which consists of 122 states and 5 processes. 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. For model checking and simulation of the PROMELA model with SPIN [Spi], 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 [VT98]. Only some PROMELA channels have to be marked as observable, viz. the ones where observable actions may occur.

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.

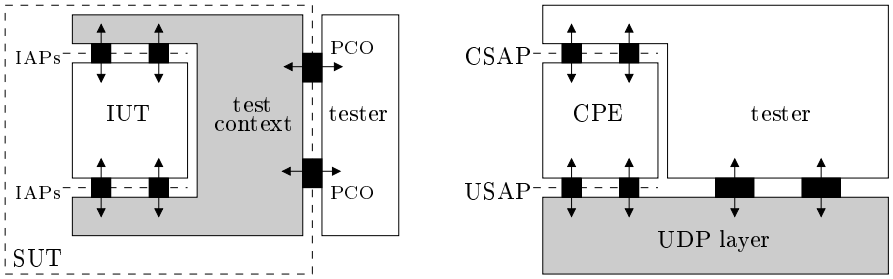
The conference protocol implementation has two interfaces: the CSAP and the USAP. The CSAP interface allows communication between the two UNIX processes, the user-interface and the CPE, and is implemented by two uni-directional pipes. The USAP interface allows communication between the CPE and the underlaying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

5.2 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) [ISO96]. 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 TORX are defined by PCOs. The SUT (*System Under Test*) consists of the IUT embedded in its test context. Figure 6(a) depicts an abstract test architecture.



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

Figure 6: 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 [TFPHT96], 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 6(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 PROMELA specification of section 5.1. 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 7 visualizes the complete formal model of the SUT.

To allow test derivation and test execution based on PROMELA, we had to make a model of the complete SUT in PROMELA. We might get such a model by extending the model of the CPE with queues that model the underlying UDP service, the pipes and the sockets. In our case we were able to make an optimization that allows removal of these queues without changing the observable behaviour of the protocol.

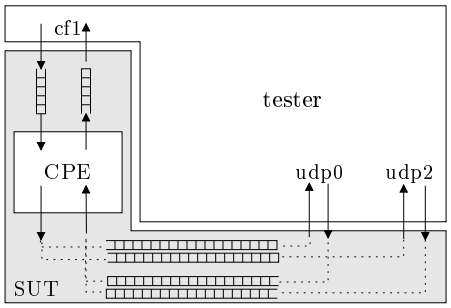


Figure 7: Formal model of the SUT

5.3 Testing Activities

This section describes our testing activities. After summarizing the overall results we will elaborate on the test activities. We used the PROMELA specification for on-the-fly test derivation and execution, using the correctness criterion **ioco**(cf. section 3). We started with initial experiments to identify errors in the specification and to test the (specification language specific) en/decoding functions of the tester. Once we had sufficient confidence in the specification and en/decoding functions, 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.

Initial experiments We started by repeatedly running TORX 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 specification 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 TORX in user-guided, manual mode to explore specific scenarios and to analyse failures that were found in fully automatic mode.

Manual guidance Figure 8 shows the graphical user interface of TORX; this is the user interface that we used for user-guided, manual mode testing. From top to bottom, the most important elements in it are the following. At the top, the *Path* pane shows the test events that have been executed so far. Below it, the *Current state offers* pane shows the possible inputs and outputs at the current state. Almost at the bottom, the *Verdict* pane will show us the verdict. Finally, the bottom pane shows diagnostic output from TORX, and diagnostic messages from the SUT.

In the *Path* pane we see the following scenario. In test event 1, the conference user (played by TORX) joins conference 52 with user-id 101, by issueing a *join* service primitive at PCO cf1. of which The SUT (at address 1) informs the two other potential conference partners at PCO udp2 (at address 2) and PCO udp0 (at address 0) in test event 2 resp. test event 3 using *join-PDUs*. The *Current state offers* pane shows the possible inputs and the expected outputs (output *Delta* means: *quiescense*, cf. section 3). The input action that we are about to select is highlighted: we will let a conference partner join a conference as well. The tester will choose values for the variables shown in the selected input event, if we don't supply values ourselves.

Figure 9 shows the TORX window after selecting the Selected Input button. Test step 4 in the *Path* pane shows that now the conference partner at PCO udp0 has joined conference 52 as well, using user-id 102, by sending a *join-PDU* (from address 0) to the SUT (at address 1). The list of *outputs* now shows that the SUT (at address 1) should respond by sending an *answer-PDU* containing its own user-id and conference-id to address 0, i.e. to PCO udp0. We will now select the Output button to ask TORX for an observation.

Figure 10 shows the TORX window after selecting the Output button to observe an output. The

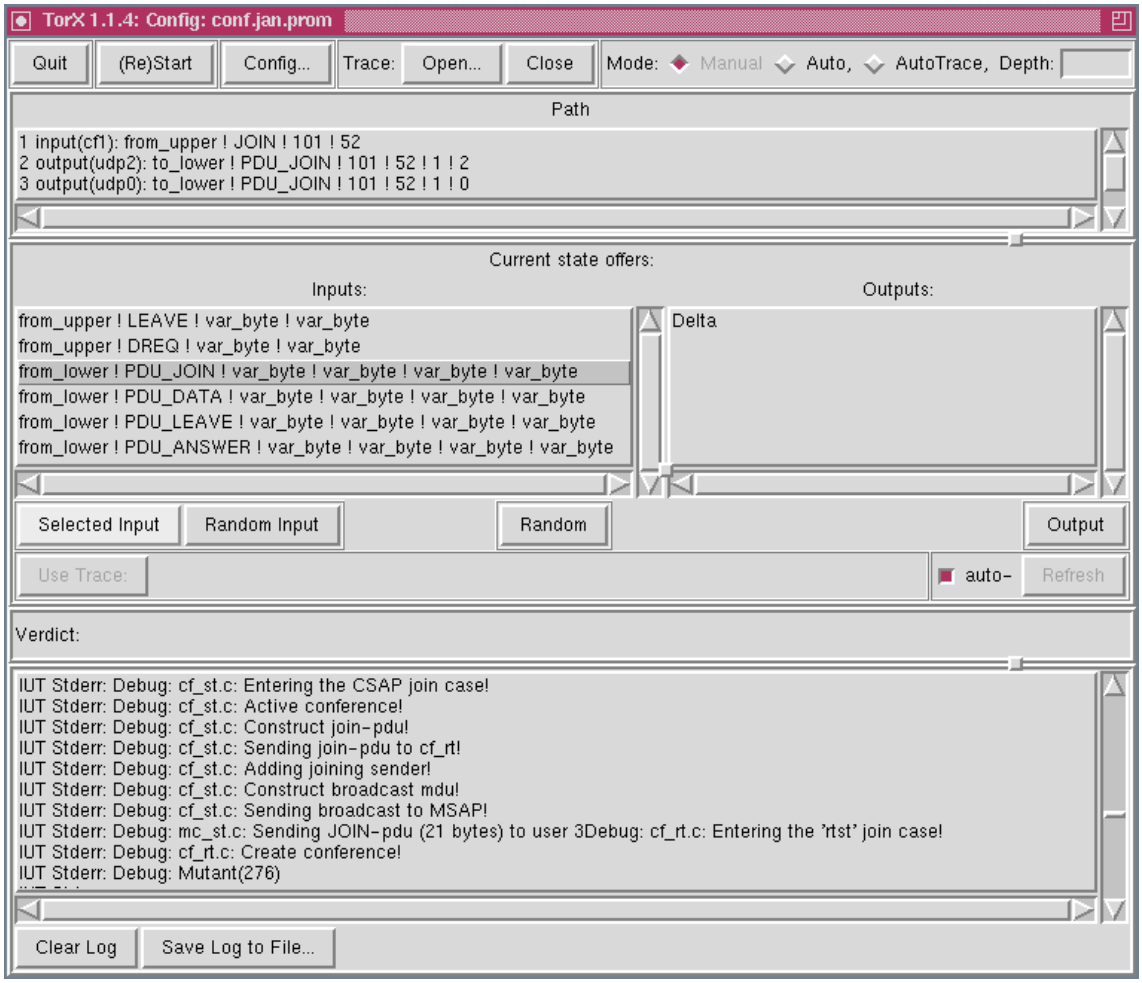


Figure 8: TorX Graphical User Interface – Selecting Input

presence of test event 5, *Quiescense*, shows that the tester did not receive anything from the SUT, which is not allowed (there is no *Delta* in the list of outputs), and therefore TorX issues a **fail** verdict. Indeed, in this example we tested one of the mutants, in particular, one that does not 'remember' that it has joined a conference, and therefore does not respond to incoming *join-PDUs*.

In a separate window, TorX keeps an up-to-date message sequence chart of the test run. This message sequence chart shows the messages interchanged between the SUT (*iut*) and each of the PCO's. A separate line represents the 'target' of Quiescense. The message sequence chart for the test run described above is shown in figure 11.

Long-running experiment Once we had sufficient confidence in the quality of the specification and implementation we repeated the previous 'automatic mode' experiment, but now we tried to execute as many test steps as possible. The longest trace we were able to execute consisted of 450,000 steps and took 400 Mb of memory. On average the execution time was about 1.1 steps per second.

Mutants detection To test the error-detection capabilities of our tester we repeatedly ran TorX 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 number of test events in the shortest test runs that detected the mutants ranged from 2 to 38; on average 18

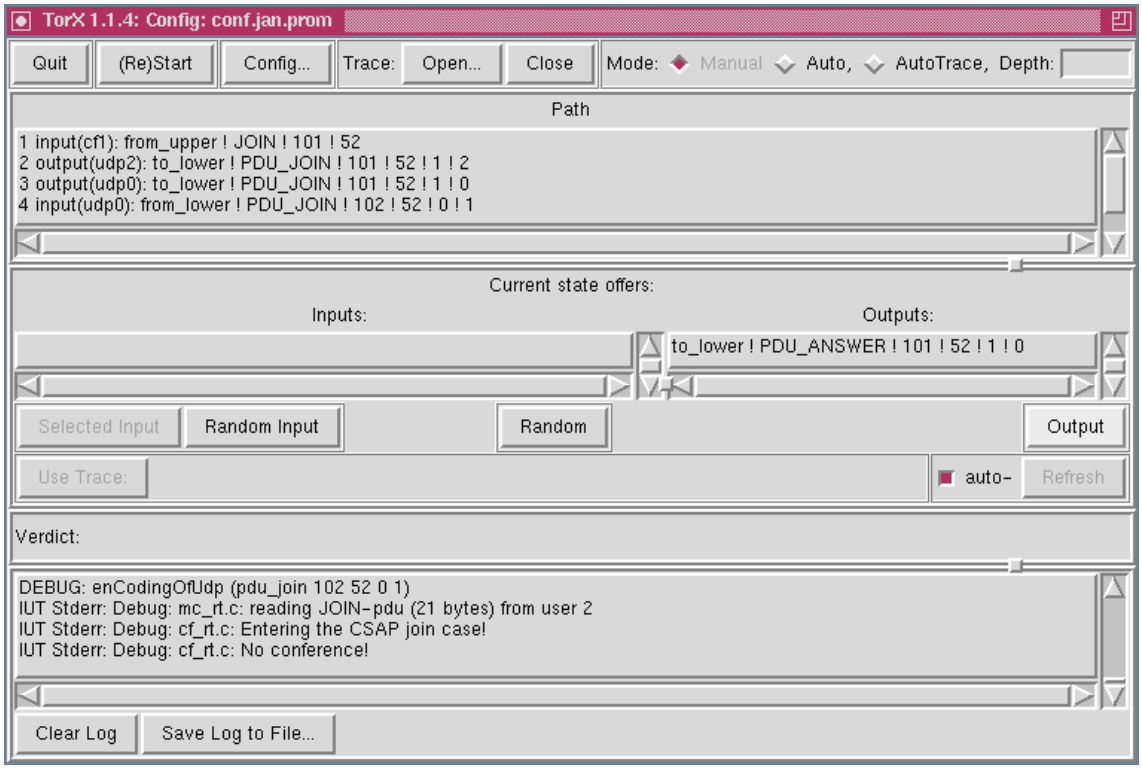


Figure 9: TorX Graphical User Interface – Selecting Output

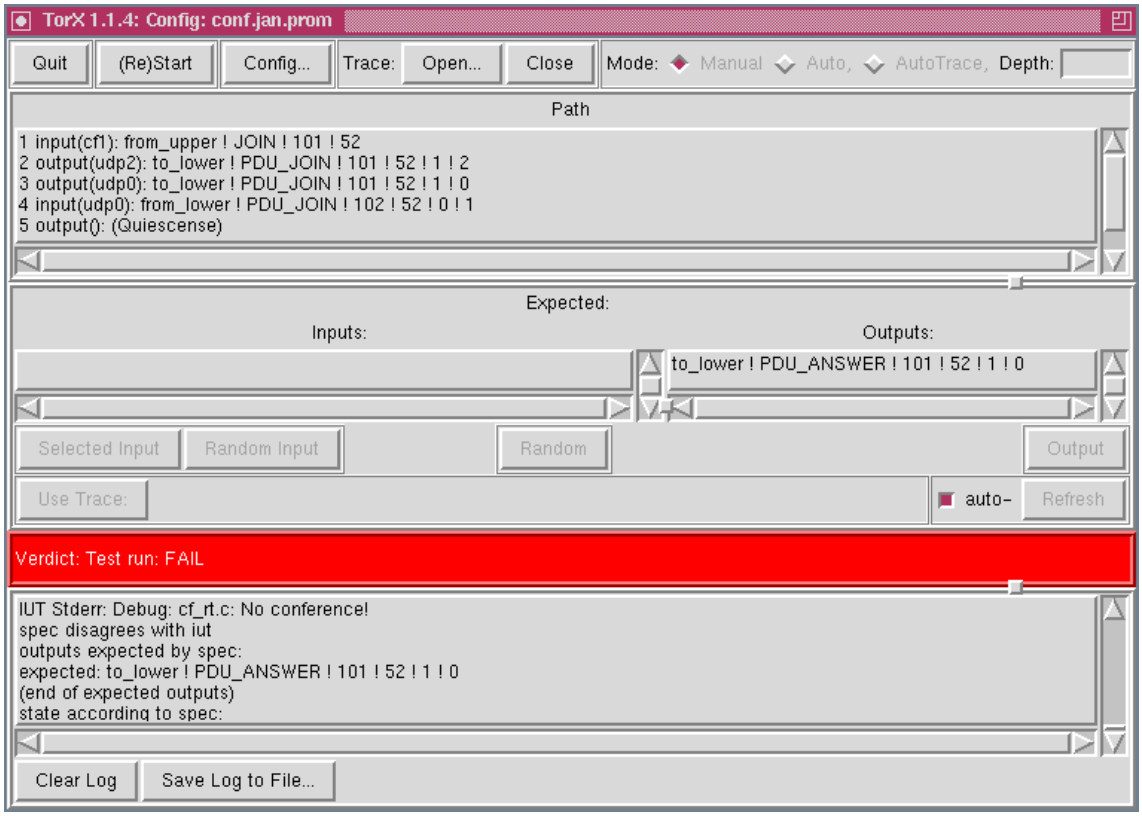


Figure 10: TorX Graphical User Interface – Final Verdict

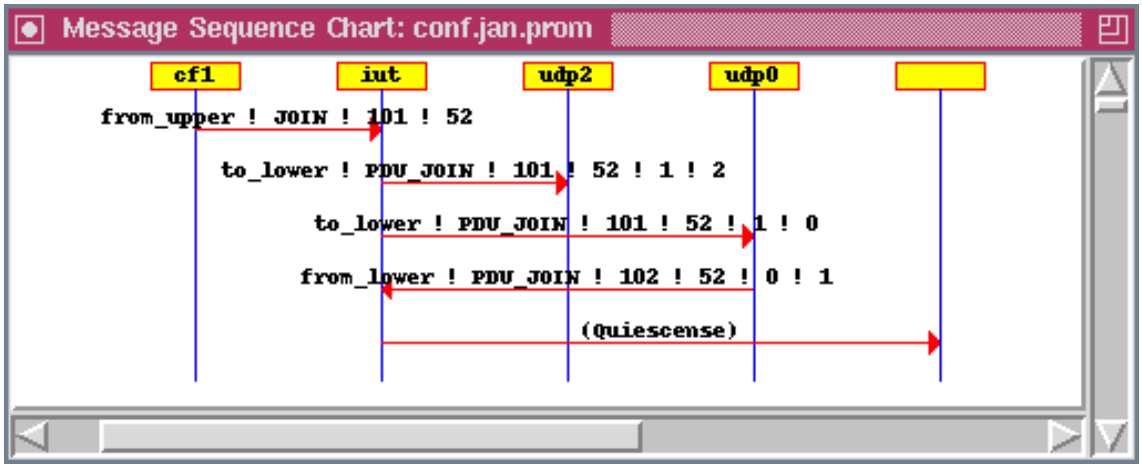


Figure 11: TorX Message Sequence Chart

test events were needed. 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 Promela specification, and therefore these mutants are **ioco**-correct with respect to the Promela specification, which is why we can not detect them.

Other formalisms We have repeated the experiments described above with specifications in LOTOS and SDL, as described in [BFV+99]. In the case of LOTOS, we used the same fully-automatic on-the-fly test-derivation and execution approach, giving us the same results as for Promela, but needing more processing time and consuming more memory. The main reason for this is that the Promela-based tester uses the memory-efficient internal data representations and hashing techniques to remember the result of unfoldings from spin. In the case of SDL, we used a user-guided test-derivation approach, after which the test-cases were automatically executed. Here we were not able to detect all mutants, but this may be due to the limited number of test-cases that we derived; by deriving more test-cases we will likely be able to detect more mutants.

Characteristics of test cases Which test cases are generated by TorX only depends on the seed of the random number generator with which TorX is started, and the (possibly non-deterministic) outputs of the SUT. From one seed to another, the number of test events needed to trigger an error may vary significantly. For the conference protocol, for one seed it took only two test events to detect a mutant, and for another seed it took 10 test events to detect the same mutant. For another mutant the differences were even more extreme: for one seed it could be detected in 24 test events, for another seed it took 498 steps. Still, as effectively in all test runs all mutants were (finally) detected, the results seem to indicate that if we run TorX sufficiently often, with varying seeds, and let it run long enough, then all errors are found. This is, unfortunately, currently also the only indication of the coverage that we have.

Logfile analysis During a test run, a log is kept to allow analysis of the test run. Such a log contains not only behavioural information to allow analysis of the test run, like tester configuration information, and the executed test events in concrete and in abstract form, but also information that allows analysis of the tester itself, like, for each test event, a timestamp, the memory usage of the computation of the test event, and a number of other statistics from the test derivation module. In case of a **fail** verdict, the expected outputs are included in the log. TorX is able to rerun a log created in a previous test run. The current TorX implementation may fail to rerun a log if the SUT behaves nondeterministically, i.e. if the order in which outputs are observed during the rerun differs from the order in which they are present in the log. This limitation may show up, for example, for the *join-PDUs* in test events 2 and 3 in figure 8.

Comparison with traditional testing In traditional testing the number of test events needed to trigger an error will quite likely be smaller than for TorX, thanks to the human guidance that will lead to 'efficient' test cases, whereas TorX may 'wander' around the error for quite a while until finally triggering it. On the other hand, to generate a new test case with TorX it is sufficient to invoke it once more with a so far untried seed, whereas approaches that are based on manual production of test cases need considerably more manual effort to derive more test cases. The main investment needed to use TorX lies in making the specification, and connecting TorX to the SUT. Once that has been arranged, the testing itself is just a case of running TorX often and long enough (and, of course, analysing the test logs).

6 Evaluation, Perspectives and Concluding Remarks

In section 5 we showed the feasibility of completely automated testing, including both test generation and test execution, based on a formal specification of system behaviour. The Conference Protocol example, however, is a rather small system, although containing some tricky details and distribution of interfaces. In this section we discuss some of the possibilities and problems in extending the results of the Conference Protocol case study.

A/V Link protocol Philips Research Laboratories in Eindhoven have started a project for automatic testing of implementations of the A/V Link protocol. The A/V Link is a protocol for communication between T.V. sets and video recorders for downloading of presettings, etc. A formal specification of A/V Link in the language Promela has been developed and a test environment interacting with T.V. sets and video recorders is now being built. Automatic testing with TorX will soon start. Comparison with the current Philips test technology implemented in the tool Phact is one of the main issues of testing the A/V Link protocol [FMMW98]. More industrial case studies are expected after concluding the A/V Link case study.

Formal specifications One of the issues prohibiting rapid introduction of automatic testing based on formal specifications is the lack of formal specifications. A formal specification of system behaviour is necessary to start automatic derivation of tests. This issue can be solved once it can be shown that the return on investment for developing a formal specification is very high in terms of a completely automated testing process. We expect that this can soon be the case for particular classes of systems, especially, since testing is usually very expensive and laborious. Automated testing will be extra beneficial if systems are changing, as they usually do. Keeping manually generated test suites up to date with changing specifications is one of the big bottlenecks of testing. When using automatic derivation of test suites, keeping them up to date is for free.

A second point to be made in formal specification development is that a formal specification in itself already results in a major increase in the quality of software, as several projects report, see e.g., [HB95, CTW99]. Last year we reported an analogous result: even without any tool support the development and use of formal specifications for testing already turned out to increase quality and to reduce cost.

Open issues There are still some important open issues in formal test theory. In the first place there is no clear strategy yet on how to derive a restricted set of test cases. Tools like TorX are able, if time would permit, to derive millions of test cases. How to make a reasonable, or even "the best" selection is still an open theoretical problem. The current brute-force approach of TorX, however, where simply as many tests as possible are randomly generated and executed, turns out to perform reasonably well in the Conference Protocol case study. The probability of missing an

error is not larger than with traditional methods. What TorX, in fact, does is replacing human, manual quality of test cases by automatic, random quantity of test cases.

Related to the above problem is the issue of *coverage* of specifications. Current coverage tools only determine code coverage i.e., *implementation coverage*, which is to be distinguished from *specification coverage*. For specification coverage there are no standard solutions available.

An important issue for test derivation tools is the issue of the so-called *state explosion problem*. In particular, in parallel and distributed systems the number of system states can be enormous, by far exceeding the number of molecules in the universe. Test derivation tools need clever and sophisticated ways to deal with this explosion of the number of states in system specifications. Techniques developed in the area of model checking are currently used, but the limits of these techniques currently restrict application to large systems.

Conclusion Sound theories and algorithms exist for automatic testing of reactive systems based on formal specifications. On-the-fly testing tools such as the prototype tool TorX, which combine automatic test derivation with test execution, are feasible and have large potential for completely automating the testing process and thus reducing its currently high cost. Apart from cost reduction the use of formal methods improves the testing process in terms of preciseness, reduced ambiguity and improved maintainability of test suites. Moreover, the incentive to develop formal specifications in itself will lead to increased quality through specifications which are more precise, more complete, more consistent, less ambiguous, and which allow formal validation and verification.

This paper has given an overview of current developments within research and development on automatic test derivation, in particular, as they occur within the project *Côte de Resyste*. Research and developments will continue aiming at making the formal testing approach better applicable by extending TorX and obviating the open issues identified above. The current status is that small-size pilot projects are considered. Within a few years we hope to be able to test realistic, industrial, medium-size systems based on their formal specifications and to automate this in such a way that people will be eager to develop the necessary formal specifications. As a side effect this may also promote the use of formal methods.

References

[BFV+99] 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, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[Bri88] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 63–74. North-Holland, 1988.

[CCI92] CCITT. *Specification and Description Language (SDL)*. Recommendation Z.100. ITU-T General Secretariat, Geneve, Switzerland, 1992.

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

[Cla96] M. Clatin. Manuel d'utilisation de TVEDA V3. Manual LAA/EIA/EVP/109, France Télécom CNET LAA/EIA/EVP, Lannion, France, 1996.

[CTW99] M. Chaudron, J. Tretmans, and K. Wijbrans. Lessons from the Application of Formal Methods to the Design of a Storm Surge Barrier Control System. In *FM'99 – World*

Congress on Formal Methods in the Development of Computing Systems. Lecture Notes in Computer Science, Springer-Verlag, 1999.

[FJJV97] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming – Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, 29(1–2):123–146, 1997.

[FMMW98] 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, editors, 11^{th} *Int. Workshop on Testing of Communicating Systems*, pages 193–210. Kluwer Academic Publishers, 1998.

[FP95] L. Ferreira Pires. Protocol implementation: Manual for practical exercises 1995/1996. Lecture notes, University of Twente, Enschede, The Netherlands, August 1995.

[Gar98] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In B. Steffen, editor, *Fourth Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, pages 68–84. Lecture Notes in Computer Science 1384, Springer-Verlag, 1998.

[Gau95] M.-C. Gaudel. Testing can be formal, too. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, pages 82–96. Lecture Notes in Computer Science 915, Springer-Verlag, 1995.

[GWT98] W. Geurts, K. Wijbrans, and J. Tretmans. Testing and formal methods — BOS project case study. In *EuroSTAR'98:* 6^{th} *European Int. Conference on Software Testing, Analysis & Review*, pages 215–229, Munich, Germany, November 30 – December 1 1998.

[HB95] M. Hinchey and J. Bowen, editors. *Applications of Formal Methods*, International Series in Computer Science. Prentice Hall, 1995.

[Hee98] L. Heerink. *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall Inc., 1991.

[ISO89] ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard IS-8807. ISO, Geneve, 1989.

[ISO91] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework*. International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.

[ISO96] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing*. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, Geneve, 1996.

[KJG99] A. Kerbrat, T. Jéron, and R. Groz. Automated Test Generation from SDL Specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99, The Next Millennium – Proceedings of the* 9^{th} *SDL Forum*, pages 135–152. Elsevier Science, 1999.

[LY96] D. Lee and M. Yannakakis. Principles and methods for testing finite state machines. *The Proceedings of the IEEE*, August 1996.

[Pha94] M. Phalippou. *Relations d'Implantation et Hypothèses de Test sur des Automates à Entrées et Sorties.* PhD thesis, L'Université de Bordeaux I, France, 1994.

[SEK+98] M. Schmitt, A. Ek, B. Koch, J. Grabowski, and D. Hogrefe. – AUTOLINK – Putting SDL-based Test Generation into Practice. In A. Petrenko and N. Yevtushenko, editors, 11^{th} *Int. Workshop on Testing of Communicating Systems*, pages 227–243. Kluwer Academic Publishers, 1998.

[Spi] Spin. On-the-Fly, LTL Model Checking with SPIN.
URL: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[Spi92] J.M. Spivey. *The Z Notation: a Reference Manual (2nd edition).* Prentice Hall, 1992.

[TFPHT96] R. Terpstra, L. Ferreira Pires, L. Heerink, and J. Tretmans. Testing theory in practice: A simple experiment. In T. Kapus and Z. Brezočnik, editors, *COST 247 Int. Workshop on Applied Formal Methods in System Design*, pages 168–183, Maribor, Slovenia, 1996. University of Maribor.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, pages 46–65. Lecture Notes in Computer Science 1664, Springer-Verlag, 1999.

[VT98] R.G. de Vries and J. Tretmans. On-the-Fly Conformance Testing using SPIN. In G. Holzmann, E. Najm, and A. Serhrouchni, editors, *Fourth Workshop on Automata Theoretic Verification with the* SPIN *Model Checker*, ENST 98 S 002, pages 115–128, Paris, France, November 2, 1998. Ecole Nationale Supérieure des Télécommunications. Also to appear in *Software Tools for Technology Transfer.*

Authors

Jan Tretmans is research associate at the University of Twente in the Formal Methods and Tools research group of the department of Computer Science. He is working in the areas of software testing and the use of formal methods in software engineering; in particular, he likes to combine these two topics: testing based on formal specifications. In this field he has several publications and he has given presentations at scientific conferences as well as for industrial audiences. Currently, Jan Tretmans is leading the project *Côte de Resyste* which is a joint industrial-academic research and development project. It addresses theory, tools and applications of conformance testing of reactive systems based on formal methods.

Axel Belinfante is research assistant at the University of Twente in the Formal Methods and Tools research group of the department of Computer Science. He is working on the development of tools and theory to support the use of formal methods in software engineering. Currently, Axel Belinfante is also involved in the project *Côte de Resyste* where he is mainly involved in test tool development – he developed the first prototype of TORX – and in issues of test execution and of encoding and decoding of abstract test events.

Acknowledgements

Our colleagues from the *Côte de Resyste* project are acknowledged for their support in the developments described in this paper: Ron Koymans and Lex Heerink from Philips Research Laboratories Eindhoven; Loe Feijs, Sjouke Mauw and Nicolae Goga from Eindhoven University of Technology; and Ed Brinksma, Jan Feenstra and René de Vries from the University of Twente.