Int J Software Informatics, Vol.XX, No.XX, XXXXXX 20XX, pp. 1–000 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org
©20XX by Institute of Software, Chinese Academy of Sciences. All rights reserved.Tel: +86-10-62661040

Fault-based Conformance Testing in Practice

Martin Weiglhofer and Bernhard Aichernig and Franz Wotawa
Institute for Software Technology - Graz University of Technology
Inffeldgasse 16b/II - 8010 Graz - Austria

Abstract Conforming to protocol specifications is a critical issue in modern distributed software systems. Nowadays, complex service infrastructures, such as Voice-over-IP systems, are usually built by combining components of different vendors. If the components do not correctly implement the various protocol specifications, failures will certainly occur. In the case of emergency calls this may be even life-threatening. Functional black-box conformance testing, where one checks the conformance of the implemented protocol to a specification becomes therefore a major issue. In this work, we report on our experiences and findings when applying fault-based conformance testing to an industrial application. Besides a discussion on modeling and simplifications we present a technique that prevents an application from implementing particular faults. Faults are modeled at the level of the specification. We show how such a technique can be adapted to specifications with large state spaces and present results obtained when applying our technique to the Session Initiation Protocol and to the Conference Protocol. Finally, we compare our results to random and scenario based testing.

Key words: fault-based testing, mutation testing, input-output conformance, ioco

1 Introduction

Validation and verification is an important task in the development of safety-critical and highly available software systems. Modern distributed systems often rely on the communication of standardized protocols implemented by different vendors. To ensure a proper cooperation of the different implementations they all must conform to the protocol specification. That is, one needs to check the conformance of the observable behavior of implementations. By considering the implementation as a black-box, i.e. without any access to the source code, one can validate the observable behavior with respect to a specification. Functional black-box testing is the method of choice for such end-to-end testing.

However, testing if conducted thoroughly and systematic is a rather tedious and time-consuming task. A formal notion of conformance and mature research prototypes suggest the application of formal methods for automatic test case generation. However, in practice there are still some challenges and issues that need to be considered:

Test case selection The early work on formal conformance testing in the area of distributed systems was mainly concerned with the soundness and completeness of the testing theory. Since the models were finite labeled transition systems, the problem of how to select a manageable subset out of the exhaustive test set was not

‡ Corresponding author: Martin Weiglhofer, Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/II, 8010 Graz, Austria, e-mail: weiglhofer@ist.tugraz.at.

a major concern. However, in practice time and resources for testing are usually very limited. Thus, there is a need for selecting a proper subset of test cases.

Identifying test purposes Test purposes, i.e. formalized test objectives, have been introduced to allow efficient test case generation for large specifications. Unfortunately, this still leaves the tester with the task of writing test purposes, which might turn out to be rather difficult if thorough testing is required. For example, du Bousquet et al. [dBRS+00] report that even after ten hours of manual test purpose design they failed to find a set of test purposes that would detect all mutants of a given implementation.

In our approach, we want to support the tester in formalizing test purposes, by turning his focus on possible faults. Possible faults can be anticipated by inspecting a specification, by using domain knowledge, or by heuristic fault injection operators. In all cases, the fault is modeled at the specification level by altering the specification syntactically. We call this altered version a mutant. The idea, is to generate test cases that would find such faults in the implementation.

Equivalent mutants A common problem with this approach is known as the Equivalent Mutant Problem. Not all mutations represent actual faults that can be observed at the interface level. Thus, no test case exists that can distinguish the original from such an equivalent mutant. On the specification level, equivalence/preorder checkers can be used to eliminate such equivalent mutants. The problem is which equivalence/preorder relation is appropriate for our purposes. Once, the relation is fixed, the problem is theoretically solved.

Applicability to industrial specifications In order to detect equivalent mutants one needs to compare the complete state spaces of the mutant and the original specification. Due to time and resource limits this is often infeasible for industrial specification. However, equivalent mutants do not lead to useful test cases because they cannot be distinguished from the specification. Test cases derived from such mutants do not improve test suites. This paper presents an approach applicable to large specifications while guaranteeing that no redundant test cases for equivalent mutants are generated.

Automated test case generation The technique should automatically generate test cases. Many use the counter examples (or witnesses) produced by a model checker as test cases. A counter example is not a test case in the traditional sense. A test case should provide the stimuli and the responses for a system. However, a counterexample exemplifies only one possible choice of computation (a path). In case of non-determinism involved this is not sufficient for a test case, since a test case should predict and take care of all possible responses, as well as reject wrong responses.

This paper presents a fully automated, fault-based, practical approach for test case selection based on LOTOS specifications. The presented approach focuses on simple faults on the specification's level. Given a specification, such faults are inserted automatically into the specification. Faulty specifications are called mutants. Note that there is only one fault per mutant. A conformance check between a faulty specification and the original specification leads to a (linear) counterexample if the mutant does not conform to the specification. Such a linear counterexample is not a valid test case for non-deterministic systems. Thus, we use this counterexample as a high-level description of the testing goal, i.e. as a test purpose. The generated test purpose allows one the use of established tools for deriving test cases based on test purposes, e.g. TGV [JJ05], SAMSTAG [GHN93], or Microsoft's SPECEXPLORER [VCG+08]. The presented approach offers the advantages of mutation based test selection strategies together with the efficiency of test purpose based test derivation algorithms.

This paper is based on our previous work [AD06, APWW07a, APWW07b, AWW08, WW08a, WW08b], which it extends with:

- Summing up our experiences of modeling and fault-based testing on industrial applications,
- a detailed discussion of formalizing the Session Initiation Protocol (SIP) in terms of a LOTOS specification,
- a new approach for handling specifications with large, industrial scale, state spaces,
- new experimental results showing the practicability when applying the presented approach to the Session Initiation Protocol,
- and results obtained by applying our fault-based testing technique to an additional protocol specification, the Conference Protocol [TPHT96].

Case Study Our main application domain is testing in the context of Voice-over-IP. One elementary protocol within the Voice-over-IP landscape is the Session Initiation Protocol.

The Session Initiation Protocol (SIP) [RSC+02] handles communication sessions between two end points. The focus of SIP is the signaling part of a communication session independent of the used media type between two end points. More precisely, SIP provides communication mechanisms for *user management* and for *session management*. *User management* comprises the determination of the location of the end system and the determination of the availability of the user. *Session management* includes the establishment of sessions, transfer of sessions, termination of sessions, and modification of session parameters. SIP defines various entities that are used within a SIP network. One of these entities is the *Registrar*, which is responsible for maintaining location information of users. An example call flow of the registration process is shown in Figure 1. In this example, Bob tries to register his current device as end point for his address Bob@home.com. Because the server needs authentication, it returns "401 Unauthorized". This message contains a digest which must be used to re-send the register request. The second request contains the digest and the user's authentication credentials, and the Registrar accepts it and answers with "200 OK". For a full description of SIP we refer to [RSC+02].

The SIP Registrar will serve as a running example in this paper in order to discuss various aspects of our approach.

This paper is organized as follows. Section 2 gives a brief overview of formal conformance testing, including an instantiation of this formal testing framework based on labeled transition systems and an introduction to input-output conformance testing. Section 3 reviews the LOTOS specification language, including an example specification based on parts of the Session Initiation Protocol. Furthermore, Section 3 discusses the

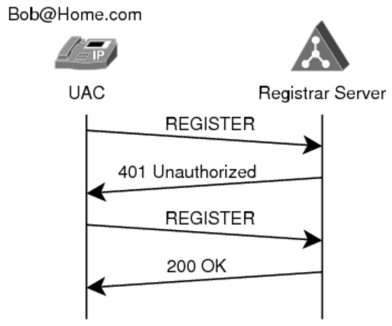


Fig. 1. Simple Call-Flow of the registration process.

need and the usage of simplifications during modeling. Section 4 shows how to apply fault-based conformance testing to industrial scale LOTOS specifications, including mutation operators for LOTOS specifications. Furthermore, this section summarizes our approach for efficient on-the-fly input-output conformance checking and extends our previous work by introducing bounded **ioco**. We applied our technique to two different specifications. Section 5 presents the obtained results and compares the approach presented in this paper with other test case selection strategies. We review related work in Section 6 and conclude in Section 7.

2 Formal Conformance Testing

An overall framework of formal conformance testing has been proposed in [IJW97, HHT96]. The central element within this conformance testing framework is the definition of what is a correct implementation of a formal specification. This defines a conformance relation. To define such conformance relations it is assumed that there exists a formal specification of the required behavior, i.e., $s \in SPECS$. The set $SPECS$ is the set of all possible specifications that can be expressed using a particular specification language.

In addition to the specification, there is the implementation under test $IUT \in IMPS$, which denotes the real, physical implementation. $IMPS$ is the universe of all possible implementations. We want to formally reason about the correctness of the concrete implementation IUT with respect to the specification s. Thus, as a common testing hypothesis [Tre92, Ber91] it is assumed that every implementation can be modeled by a formal object $m_{IUT} \in MODS$. $MODS$ denotes the set of all models. Note that it is not assumed that this model is known, only its existence is required.

Conformance is expressed as a relation between formal models of implementations and specifications, i.e.,

$$\mathbf{imp} \subseteq MODS \times SPECS$$

As each model $m_{IUT} \in MODS$ represents a concrete implementation $iut \in IMPS$ a conformance relation **imp** allows one to formally reason about the correctness of the iut with respect to a specification $s \in SPECS$.

By applying inputs to the implementation and observing outputs, i.e. by testing, one wants to find non-conforming implementations. The universe of test cases is given by $TESTS$. Executing a test case $t \in TESTS$ on an implementation leads to observations $obs \subseteq OBS$, where OBS denotes the universe of observations.

Formally, test case execution is modeled by a function $exec : TESTS \times MODS \rightarrow OBS$. Given a test case $t \in TESTS$ and a model of an implementation $m \in MODS$, $exec(t, m)$ gives the observations in OBS that result from executing t on the model m.

Finally, there is a function $verd$ that assigns a verdict, i.e. pass or fail, to each observation: $verd : OBS \rightarrow \{\mathbf{pass}, \mathbf{fail}\}$. An implementation $IUT \in IMPS$ *passes* a test suite $TS \subseteq TESTS$ if test execution of all its test cases leads to an observation for which $verd$ evaluates to **pass**. In practice, there is a third verdict, i.e. inconclusive, that is used for judging test executions [JJ05]. This verdict is used if the implementation has not done anything wrong but the responses of the IUT did not satisfy the test objective.

There are different types of models and conformance relations that can be seen as an instantiation of this formal conformance testing framework. For example, when one uses testing techniques based on finite state machines (FSMs) [LY96, HBH08] then $MODS$ and $SPECS$ are usually some sorts of FSMs. Usually, the considered conformance relation is some relation between the states of the implementation and the states of the specification (e.g. isomorphism).

Another instantiation of this formal testing framework uses labeled transition systems for representing specifications and models of implementations. There is a broad range of relations that have been defined for labeled transition systems, e.g. bisimulation equivalence [Mil90], failure equivalence and preorder [Hoa85], and refusal testing [Phi87], just to name a few.

One commonly used conformance relation is the input-output conformance relation (**ioco**) [Tre96]. This relation is designed for functional black box testing of systems with inputs and outputs. Inputs are under the control of the environment, i.e. the tester, while outputs are under the control of the implementation under test. **ioco** allows one to use incomplete specifications. The specifications and the implementations can be non-deterministic. Furthermore, the models used for **ioco** allow arbitrary interleaving of input and output. Finally, **ioco** considers the absence of outputs as error if this behavior is not allowed by the specification. These properties make input-output conformance testing applicable to practical applications.

2.1 Input-Output Conformance

The input-output conformance (**ioco**) relation [Tre96] expresses the conformance of implementations to their specifications where both are represented as labeled transition systems (LTS). Because we distinguish between inputs and outputs, the alphabet of an LTS is partitioned into inputs and outputs.

Definition 1 (LTS with inputs and outputs). A labeled transition system with inputs and outputs is a tuple $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$, where Q is a countable, non-empty set of states, $L = L_I \cup L_U$ a finite alphabet, partitioned into two disjoint sets, where L_I and L_U are input and output alphabets, respectively. $\tau \notin L$ is an unobservable action, $\rightarrow \subseteq Q \times (L \cup \{\tau\}) \times Q$ is the transition relation, and $q_0 \in Q$ is the initial state. □

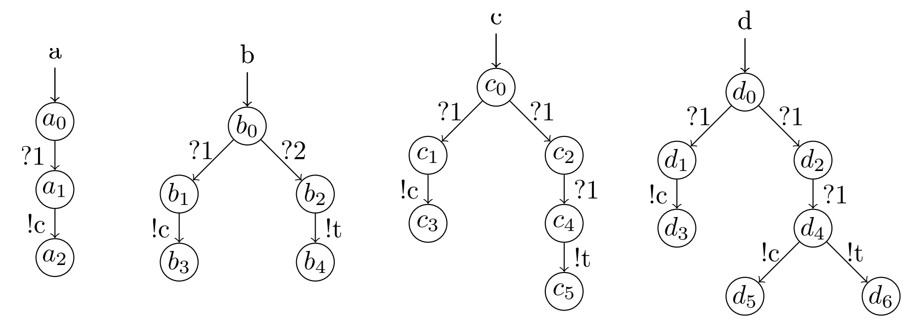


Fig. 2. Four labeled transition systems.

An LTS is *deterministic* if for any sequence of actions starting at the initial state there is at most one successor state.

We use the following common notations:

Definition 2. Given a labeled transition system $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$ and let q, q', $q_i \in Q, a_{(i)} \in L$ and $\sigma \in L^*$.

$$\begin{aligned}
q \xrightarrow{a} q' &=_{df} (q, a, q') \in \rightarrow \\
q \xrightarrow{a} &=_{df} \exists q' \bullet (q, a, q') \in \rightarrow \\
q \overset{a}{\not\rightarrow} &=_{df} \nexists q' \bullet (q, a, q') \in \rightarrow \\
q \overset{\epsilon}{\Rightarrow} q' &=_{df} (q = q') \vee \exists q_0, \dots, q_n \bullet (q = q_0 \xrightarrow{\tau} q_1 \wedge \cdots \wedge q_{n-1} \xrightarrow{\tau} q_n = q') \\
q \overset{a}{\Rightarrow} q' &=_{df} \exists q_1, q_2 \bullet q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Rightarrow} q' \\
q \overset{a_1 \dots a_n}{\Rightarrow} q' &=_{df} \exists q_0, \dots, q_n \bullet q = q_0 \overset{a_1}{\Rightarrow} q_1 \dots q_{n-1} \overset{a_n}{\Rightarrow} q_n = q' \\
q \overset{\sigma}{\Rightarrow} &=_{df} \exists q' \bullet q \overset{\sigma}{\Rightarrow} q'
\end{aligned}$$

□

We use $init(q)$ to denote the actions enabled in state q. Furthermore, we denote the set of states reachable by a particular trace σ by q **after** σ. More precisely,

Definition 3. Given an LTS $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$ and $q \in Q, S \subseteq Q$, $a \in L$, and $\sigma \in L^*$:

$$\begin{aligned}
init(q) &=_{df} \{a | q \xrightarrow{a}\} \\
init(S) &=_{df} \bigcup_{q \in S} init(q) \\
q \textbf{ after } \sigma &=_{df} \{q' | q \overset{\sigma}{\Rightarrow} q'\} \\
S \textbf{ after } \sigma &=_{df} \bigcup_{q \in S} (q \textbf{ after } \sigma)
\end{aligned}$$

□

Note that we will not always distinguish between an LTS M and its initial state and write $M \Rightarrow$ instead of $q_0 \Rightarrow$.

Example 1. Figure 2 shows four labeled transition systems representing a coffee/tee (c/t) vending machine. The input and output alphabets are given by $L_I = \{1, 2\}$ and by $L_U = \{c, t\}$, i.e. one can insert one and two unit coins and the machine outputs

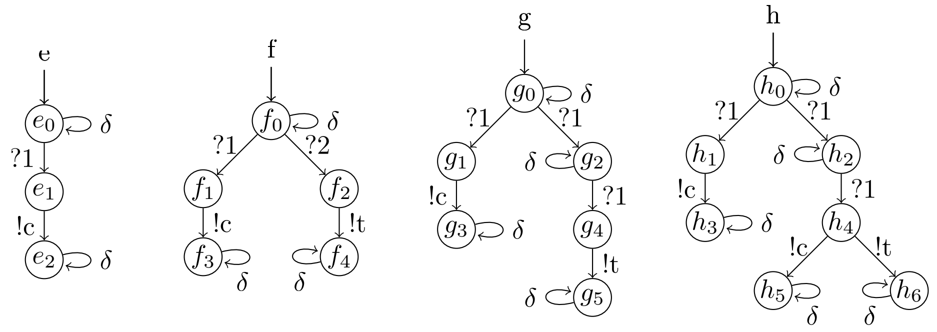


Fig. 3. δ-annotated labeled transition systems.

coffee or tea. We denote input actions by the prefix "?", while output actions have the prefix "!". For example, a_0 **after** $\langle ?1\rangle = \{a_1\}$ while c_0 **after** $\langle ?1\rangle = \{c_1, c_2\}$. □

The **ioco** conformance relation employs the idea of observable quiescence. That is, it is assumed that a special action, i.e. δ, is enabled in the case where the labeled transition system does not provide any output action. These δ-labeled transitions allow to detect implementations that do not provide outputs while the specification requires some output (see Example 4: $\neg(k$ **ioco** $e)$). The input output conformance relation identifies quiescent states as follows: A state q of a labeled transition system is quiescent if neither an output action nor an internal action (τ) is enabled in q.

Definition 4. Let M be a labeled transition system $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$, with $L = L_I \cup L_U$, such that $L_I \cap L_U = \emptyset$, then a state $q \in Q$ is quiescent, denoted by $\delta(q)$, if $\forall a \in L_U \cup \{\tau\} \bullet q \overset{a}{\not\rightarrow}$. □

By adding δ-labeled transitions to LTSs the quiescence symbol can be used as any other action.

Definition 5. Let $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$ be an LTS then $M_\delta = (Q, L \cup \{\tau, \delta\}, \rightarrow \cup \rightarrow_\delta, q_0)$ where $\rightarrow_\delta =_{df} \{q \xrightarrow{\delta} q | q \in Q \wedge \delta(q)\}$. The suspension traces of M_δ are $Straces(M_\delta) =_{df} \{\sigma \in (L \cup \{\delta\})^* | q_0 \overset{\sigma}{\Longrightarrow}\}$. □

Unless otherwise indicated, from now on we include δ in the transition relation of LTSs, i.e., we use M_δ instead of M.

The class of labeled transition systems with inputs L_I and outputs in L_U (and with quiescence) is denoted by $\mathcal{LTS}(L_I, L_U)$ [Tre96]. This is the universe of specifications, i.e. $SPECS = \mathcal{LTS}(L_I, L_U)$.

Example 2. Figure 3 shows the δ-annotated LTSs for the LTSs illustrated in Figure 2. For example, the states g_0, g_2, g_3, and g_5 are quiescent because they do not have outputs nor τ actions. □

Models for implementations in terms of the input-output conformance relation are input-output transition systems (IOTS). Recall that it is not assumed that this LTS is known in advance, but only its existence is required. Implementations are not allowed to refuse inputs, i.e. implementations are assumed to be input-enabled and so are their models. Note that specifications do not have to be input-enabled.

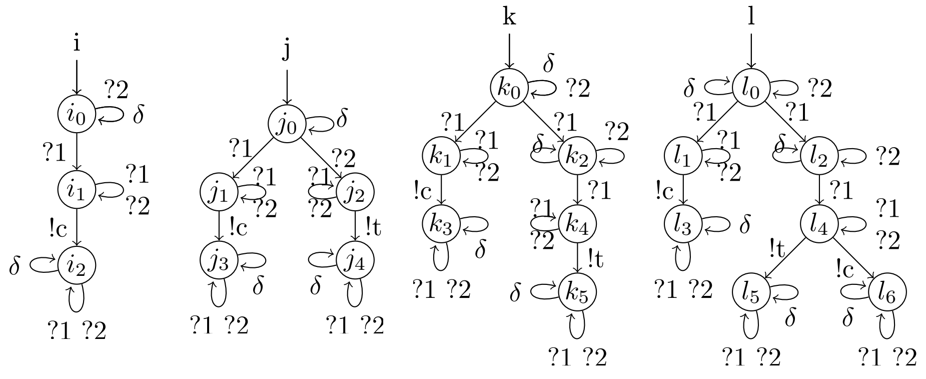


Fig. 4. Input-output transition systems.

Definition 6 (IOTS). An input-output transition system is an LTS $M = (Q, L \cup \{\tau\}, \rightarrow, q_0)$, with $L = L_I \cup L_U$, such that $L_I \cap L_U = \emptyset$, where all input actions are enabled (possibly preceded by τ-transitions) in all states: $\forall a \in L_I, \forall q \in Q \bullet q \overset{a}{\Longrightarrow}$. □

Note that this sort of input-enabledness, i.e. where τ-labeled transitions may precede input actions ($\forall a \in L_I, \forall q \in Q \bullet q \overset{a}{\Longrightarrow}$), is called *weak input-enabledness.* Contrary, strong input-enabledness requires that all input actions are enabled in all states, i.e. $\forall a \in L_I, \forall q \in Q \bullet q \xrightarrow{a}$.

The class of IOTSs with inputs L_I and outputs in L_U is given by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$ [Tre96]. As IOTSs are used to formally reason about implementations, input-output transition systems are our implementation models, i.e. $MODS = \mathcal{IOTS}(L_I, L_U)$ when instantiating the formal framework of conformance testing.

Example 3. Figure 4 depicts the IOTSs derived from the LTSs of Figure 3. □

We use $out(q)$ to denote the outputs at a state q.

Definition 7. Given a labeled transition system $M = (Q, L \cup \{\tau, \delta\}, \rightarrow, q_0)$, with $L = L_I \cup L_U$, such that $L_I \cap L_U = \emptyset$, let $q \in Q$ and $S \subseteq Q$, then

$$out(q) =_{df} \{a \in L_U |\; q \xrightarrow{a}\} \cup \{\delta | \delta(q)\}$$
$$out(S) =_{df} \bigcup_{q \in S} (out(q))$$

□

Informally, the input-output conformance relation states that an implementation under test (IUT) conforms to a specification S iff the outputs of the IUT are outputs of S after an arbitrary suspension trace of S. Formally, **ioco** is defined as follows:

Definition 8 (Input-output conformance). Given a set of inputs L_I and a set of outputs L_U then $\textbf{ioco} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I, L_U)$ is defined as:

$$IUT \;\textbf{ioco}\; S =_{df} \forall \sigma \in Straces(S) \bullet out(IUT \;\textbf{after}\; \sigma) \subseteq out(S \;\textbf{after}\; \sigma)$$ □

Example 4. Consider the LTSs of Figure 3 to be specifications and let the IOTSs of Figure 4 be implementations. Then we have i **ioco** e and j **ioco** f. We also have j **ioco** e because $\langle ?2 \rangle$ is not a trace of e. Thus, this branch is not relevant with respect

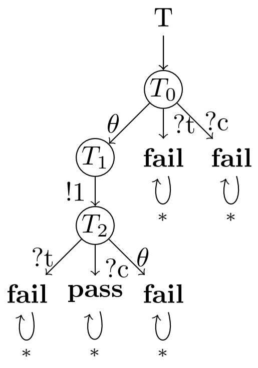


Fig. 5. Example of a test case.

to **ioco**. k does not conform to e, i.e. $\neg(k$ **ioco** $e)$, because $out(k$ **after** $?1) = \{!c, \delta\} \not\subseteq \{!c\} = out(e$ **after** $?1)$. Furthermore $\neg(l$ **ioco** $e)$ because $out(l$ **after** $?1) = \{!c, \delta\} \not\subseteq \{!c\} = out(e$ **after** $?1)$. Due to the use of suspension traces we also have $\neg(l$ **ioco** $k)$ because $out(l$ **after** $\langle ?1, \delta, ?1\rangle) = \{!c, !t\} \not\subseteq \{!t\} = out(k$ **after** $\langle ?1, \delta, ?1\rangle)$. □

2.2 Test Cases and Test Case Execution

By the use of a particular set of test cases one wants to test if a given implementation conforms to its specification. In the **ioco** framework a test case is a labeled transition system [Tre96]. In a test case the observation of δ is implemented by θ, i.e., test cases use θ, to observe δ. This is because, in practice δ is a timeout, which is not a normal event that can be observed. θ can be seen as the timer used to observe the occurrence of quiescence, i.e. the occurrence of δ. Note that inputs of a test case are outputs of an IUT and vice versa.

Definition 9 (Test case). A test case T is an LTS $T = (Q, L \cup \{\theta\}, \rightarrow, q_0)$, with $L = L_I \cup L_U$, and $L_I \cap L_U = \emptyset$, such that (1) T is deterministic and has finite behavior; (2) Q contains sink states **pass** and **fail** (in **pass** and **fail** states a test case synchronizes on any action); and (3) for any state $q \in Q$ where $q \neq$ **pass** and $q \neq$ **fail**, either $init(q) = \{a\}$ for some $a \in L_U$, or $init(q) = L_I \cup \{\theta\}$. □

$\mathcal{TEST}(L_U, L_I)$ denotes the class of test cases over L_U and L_I, i.e. $TESTS = \mathcal{TEST}(L_U, L_I)$. A *test suite* is a set of test cases.

Example 5. Figure 5 shows a test case satisfying Definition 9. In a state either inputs (outputs of the IUT) and the θ event are enabled, or an output is enabled. The $*$-labeled transitions in **pass** and **fail** states denote that in such states a test case synchronizes on any actions. □

Running a test case $t \in \mathcal{TEST}(L_U, L_I)$ against an implementation under test $i \in \mathcal{IOTS}(L_I, L_U)$ is similar to the parallel composition of the test case and the implementation. The only difference is that θ is used to observe δ. Formally, running t on i is denoted by $t\rceil|i$.

Definition 10 (Synchronous execution). Given a test case $t \in \mathcal{TEST}(L_U, L_I)$, an IUT $i \in \mathcal{IOTS}(L_I, L_U)$, and let $a \in L_U \cup L_I$, then the synchronous test case execution operator $\rceil|$ has the operational semantics defined by the following inference rules:

$$\frac{i \xrightarrow{\tau} i'}{t\rceil|i \xrightarrow{\tau} t\rceil|i'} \qquad \frac{t \xrightarrow{a} t',\ i \xrightarrow{a} i'}{t\rceil|i \xrightarrow{a} t'\rceil|i'} \qquad \frac{t \xrightarrow{\theta} t',\ i \xrightarrow{\delta}}{t\rceil|i \xrightarrow{\theta} t'\rceil|i}$$ □

A test run can always continue, i.e. there are no deadlocks. This is because a test case synchronizes on any action when a verdict state is reached. Formally, a test run is a trace of $t\rceil|i$ leading to a verdict state (**pass**, **fail**) of t:

Definition 11 (Test run). Given a test case $t \in \mathcal{TEST}(L_U, L_I)$ and an IUT $i \in \mathcal{IOTS}(L_I, L_U)$, then a test run $\sigma \in L_U \cup L_I \cup \{\theta\}$ is given by: $\exists i' \bullet t\rceil|i \overset{\sigma}{\Longrightarrow} \mathbf{pass}\rceil|i'$ or $\exists i' \bullet t\rceil|i \overset{\sigma}{\Longrightarrow} \mathbf{fail}\rceil|i'$. □

An implementation i passes a test case iff all possible test runs lead to **pass** verdict states of the test case:

Definition 12 (Passing). Given an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ and a test case $t \in \mathcal{TEST}(L_U, L_I)$, then i **passes** $t \Leftrightarrow_{df} \forall \sigma \in (L_I \cup L_U \cup \theta)^*, \forall i' \bullet t\rceil|i \overset{\sigma}{\not\Longrightarrow}$ $\mathbf{fail}\rceil|i'$ □

Example 6. Running the test case of Figure 5 on the IUT i of Figure 4 leads to the following test runs:

$$T_0\rceil|i_0 \overset{\langle\theta,!1,?c\rangle}{\Longrightarrow} \mathbf{pass}\rceil|i_2$$
$$T_0\rceil|i_0 \overset{\langle\theta,!1,?c,\theta\rangle}{\Longrightarrow} \mathbf{pass}\rceil|i_2$$
$$T_0\rceil|i_0 \overset{\langle\theta,!1,?c,\theta,\theta\rangle}{\Longrightarrow} \mathbf{pass}\rceil|i_2$$
$$\ldots$$

Because all possible test runs lead to pass, we have i **passes** T. □

Due to the structure of test cases, a test case may block outputs of an IUT. If a test case likes to provide a stimuli (input to the implementation) but the implementation opts for an output, the test case rules [PY02]. To overcome this issue, test cases have been made input-enabled recently [Tre08]. Input-enabled test cases do not block outputs of IUTs. However, such test cases comprise non-deterministic choices between inputs and outputs.

2.3 Test Case Generation

Among others, there are two test case generation strategies that have turned out to be well applicable in practice. First, there is the approach of selecting test cases randomly [Tre96]. Second, test cases are selected based on so called test purposes. A test purpose can be seen as a formal specification of a test case. Tools like SAMSTAG [GHN93], TGV [JJ05] and Microsoft's SpecExplorer [VCG$^+$08] use test purposes for test generation. Our approach relies on TGV, where test purposes are defined as LTSs:

Definition 13 (Test Purpose). Given a specification S in the form of an LTS, a test purpose is a deterministic LTS $TP = (Q, L, \rightarrow, q_0)$, equipped with two sets of trap states: $Accept^{TP}$ defines pass verdicts, and $Refuse^{TP}$ limits the exploration of the graph S. Furthermore, TP is complete (i.e., it allows all actions in each state). □

According to [JJ05] test synthesis within TGV is conducted as follows: Given a test purpose TP and a specification S, TGV calculates the synchronous product $SP = S \times TP$. The construction of SP is stopped in *Accept* and *Refuse* states as subsequent behaviors are not relevant to the test purpose. Then TGV marks all states where neither an output nor a τ-labeled transition is possible by adding δ labeled self-loops to these states (c.f. Definition 4). Before a test case is extracted, TGV obtains the observable behavior SP^{VIS} by making SP deterministic. Note that SP^{VIS} does not contain any τ-labeled transitions.

A test case derived by TGV is controllable, i.e., it does not have to choose between sending different stimuli or between waiting for responses and sending stimuli. This is achieved by selecting traces from SP^{VIS} that lead to *Accept* states and pruning edges that violate the controllability property. Finally, the states of the test case are annotated with the verdicts pass, fail and inconclusive. Inconclusive verdicts denote that neither a pass nor a fail verdict has been reached but the implementation has chosen a trace that is not included in the traces selected by the test purpose.

Although test purposes are complete, i.e. they allow actions in each state, the derived test cases satisfy Definition 9. That is, a test case either provides a stimulus to the implementation or it accepts all possible responses from the implementation under test.

As a major strength of TGV, the test case synthesis is conducted on-the-fly: parts of S, SP, and SP^{VIS} are constructed only when needed. In practice, this allows one to apply TGV to large specifications.

3 Modeling using LOTOS

Labeled transition systems are suitable models for representing test purposes, test cases, specifications, and implementations. However, for modeling large industrial systems it is impractical to write specifications in terms of labeled transition systems. Therefore, a specification language is needed having the semantics of labeled transition systems, but providing a simple syntax for writing large specifications. One such specification language is LOTOS, the language of temporal ordering specification [ISO89].

LOTOS is an ISO standard[ISO89]. LOTOS comprises two components: The first is based on the Calculus of Communication Systems [Mil80] (CCS) and deals with the behavioral description of a system, which is given in terms of processes, their behavior, and their interactions. The second component of LOTOS specifications is used to describe data structures and value expressions, and is based on the abstract data type language ACT ONE [EFH83].

The basic elements of a LOTOS specification are processes with a certain behavior expressed in terms of actions. An action is an expression over a process's gates, possibly equipped with values. Table 1 lists some of the main elements used to compose the behavior of a process. As listed in this table, an action is basically one of four expressions. There is the internal action i. There is the action without any parameters,

Table 1. Excerpt of LOTOS behavior elements.

Syntax	Meaning
`i`	internal action
`g`	action, i.e. a process gate
`g !value`	action offering a value
`g ?value:Type`	action reading a value
`action; behavior`	action followed by a behavior
`[guard] -> behavior`	guarded behavior
`behavior1 [] behavior2`	choice
`behavior1 || behavior2`	synchronization
`behavior1 ||| behavior2`	interleaving
`behavior1 |[gate1,..]| behavior2`	partial synchronization
`behavior1 [> behavior2`	disabled by second behavior
`behavior1 >> behavior2`	first enables second
`exit`	exit
`stop`	stop, inaction
`proc[gate,..](val,..)`	process instantiation
`( behavior )`	grouping

i.e. only the name of a process' gate. This simply expresses that the process offers this action for communication. Actions may offer values, i.e. `g !value` or actions may read values, i.e. `g ?value`.

Sequential composition is denoted by `action; behavior`, i.e. first the action is offered for communication and then the behavior is executed. The actions within a guarded behavior are only enabled if the guard evaluates to true.

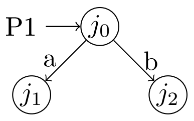
A choice between two behaviors (`behavior1 [] behavior2`) expresses that the very first action of `behavior1` and of `behavior2` are offered for communication. Once, one of the offered actions is chosen by the environment the composed process behaves like `behavior1` or `behavior2`. Basically, `[]` expresses external choice. Internal choice, i.e., a choice where a process itself decides on the offered actions, can be implemented using the special action i. i represents an internal transition, i.e. i results in a τ-labeled transition within the underlying LTS semantics.

Example 7. Figure 6 illustrates the difference between internal and external choice in LOTOS and in the underlying labeled transition systems. The specification and its LTS shown on the left depict an external choice between the actions a and b. That is, in state j_0 the environment can choose whether to synchronize on a or on b. On the contrary, the process `P2` and its LTS depict an internal choice between a and b. That is, the system may internally decide to move to state k_1 or to state k_2. In any case only one action is offered for communication. □

LOTOS supports three different operators to express the parallel composition between processes, i.e., `||`, `|||`, and `|[...]|`. The expression `P1||P2` denotes the full synchronization on all gates of the two composed processes `P1` and `P2`. The behavior of `P1|||P2` is given by the unsynchronized interleaving of the behavior of `P1` and of

```
process P1[a, b] : exit :=
  a; exit [] b; exit
endproc
```

```
process P2[a, b] : exit :=
  i; a; exit [] i; b; exit
endproc
```



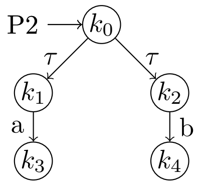


Fig. 6. External and internal choice.

the behavior of P2. Finally, `P1|[...]|P2` is used to synchronize on a particular subset of the gates of the two processes. Note that this operator can be used to express the former two.

The enabling operator (`>>`) within a LOTOS specification states that a successful execution of the first process enables the following behavior block. In contrast to that, the disabling operator (`[>`) states that any action from the second behavior disables the execution of the first behavior. Finally, the behavioral part of LOTOS supports the definition of processes (for an example see Figure 6), instantiation of processes and grouping of behavior.

An abstract data type is given in terms of sorts, operations, and equations.

Example 8. Figure 7 shows the basic elements of data type definitions. This `Configuration` data type is part of our Session Initiation Protocol Registrar specification [Wei06] and is used to represent the list of configured users. Entries in this list are UserRecord-elements. A UserRecord is an abstract type representing a tuple. The tuple comprises the user identifier (UserId) and a Boolean flag indicating the authorization status of the user. If the flag is true, the user is allowed to modify the entries stored within the SIP Registrar.

The `Configuration` data type comprises the basic constructor `nilCfg` (Line 5), denoting an empty list, and the three operators `addCfgEntry`, `getCfgEntry`, and `isin`. `isin` is an infix operator while the other two operations use a prefix notation.

For example, the signature of the `isin` operator in Line 8 declares that this operator takes a user identifier (UserId) and a configuration list (Cfg) and returns a Boolean. The axioms of the `isin` operation, with the equations identifying Boolean terms (`ofsort Bool`), are read as follows: No element is in the empty list (Line 12). If the given user identifier (uid) is equal to the user identifier (Line 13) of an element `elem`, then `uid isin addCfgEntry(elem, tail)` is true for any remaining list `tail`. If the given uid is different to the identifier of the element `elem`, then the result of `uid isin addCfgEntry(elem, tail)` is true if uid is in the rest of the list, i.e. `uid isin tail`, and false otherwise (Line 16). □

3.1 Modeling the Session Initiation Protocol in LOTOS

In this section we present parts of our Session Initiation Protocol (SIP) Registrar specification [Wei06] in order to exemplify the use of LOTOS.

A SIP Registrar provides its functionality through the maintenance of a state

```
type Configuration is UserRecord, Boolean
  sorts
     Cfg
  opns
     nilCfg : -> Cfg
     addCfgEntry : URec, Cfg -> Cfg
     getCfgEntry    : UserId, Cfg -> URec
     _isin_ : UserId, Cfg -> Bool
  eqns
     forall uid: UserId, elem: URec, tail: Cfg
        ofsort Bool
           uid isin nilCfg = false;
           (uid eq getUId(elem)) =>
             (uid isin addCfgEntry(elem, tail)) = true;
           (uid ne getUId(elem)) =>
             (uid isin addCfgEntry(elem, tail)) = (uid isin tail);
        ofsort URec
           getCfgEntry(uid, nilCfg) = newURec(2 of UserId, true);
           uid eq getUId(elem) =>
             getCfgEntry(uid, addCfgEntry(elem, tail)) = elem;
           uid ne getUId(elem) =>
             getCfgEntry(uid, addCfgEntry(elem, tail)) =
               getCfgEntry(uid, tail);
endtype (* Cfg *)
```

Fig. 7. A LOTOS abstract data type definition representing a list of configuration entries for our specification of a Session Initiation Protocol Registrar.

machine. Basically, this state machine is responsible for retransmission of responses of the SIP Registrar. Figure 8 shows the so called non-invite server transaction of a SIP Registrar. We abstracted from nodes and edges drawn in gray since they are not relevant or hard to consider within a model suited for testing this protocol.

Each REGISTER request is processed by its own state machine. As illustrated by Figure 8, an initial request is handed over to the transaction's user (TU), i.e. the Registrar core. Based on the header fields of the request the Registrar core determines a proper response and forwards the response to the state machine. Responses in the Session Initiation Protocol are identified by three digits. Since a Registrar never generates `1xx` responses, our model does not include the gray edge from the *Trying*-state to the *Proceeding*-state. Any other response, i.e.200-699, is forwarded to the initiator of the request (*send response*). The state machine then goes to its *Completed*-state.

Once, the *Completed*-state is reached any request (which matches the transaction handled by this state machine) is answered with the last sent response (self-loop on the state *Completed*). After a particular amount of time, the state machine moves to the *Terminated*-state and the transaction is destroyed.

Figure 9 shows our LOTOS formalization of the transaction handling state machine. The state machine is represented by a single process (`serverTransaction`). This process communicates with the transaction user, i.e. the SIP Registrar core, through the gates `from_tu` and `to_tu`. The gates `pin` and `pout` are used for communication with the environment, i.e. to receive REGISTER requests and to send proper responses.

Since LOTOS does not include state-variables, the state machine is implemented as a recursive process where the current state is maintained in a parameter of this process.

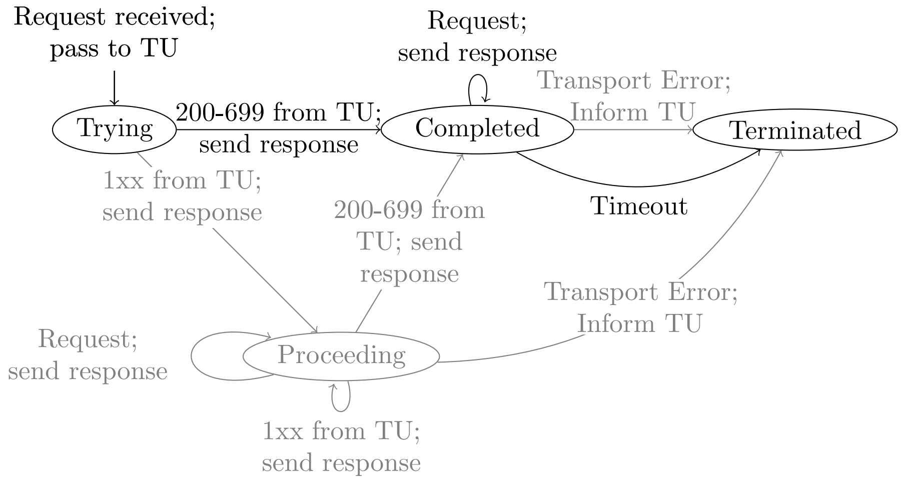

Request received; pass to TU
Request; send response
Trying
200-699 from TU; send response
Completed
Transport Error; Inform TU
Terminated
1xx from TU; send response
200-699 from TU; send response
Timeout
Transport Error; Inform TU
Request; send response
Proceeding
1xx from TU; send response

Fig. 8. State machine for handling transactions of a Session Initiation Protocol Registrar (Source: [RSC$^+$02]).

Every time the process is invoked it checks its current state and reacts according to Figure 8.

In addition to the current state parameter `trans_state` (Line 2), which may have the value `ts_trying`, `ts_completed`, or `ts_terminated`, the process takes two further parameters: `branch` (Line 3), which is used to identify retransmissions; `response` (Line 4), which holds the last sent response.

As indicated by the `noexit` keyword (Line 4), this server process does not terminate, but continues forever. Thus, as an abstraction we can only have one transaction state machine. The state machine is never destroyed, but only the relevant parameters are reset once the terminated state is entered.

If the model of the state machine is in the *Terminated*-state, it waits to receive a message and moves on to the *Trying* state if a message is received. The reception of REGISTER messages (and some initial validations) is modeled by the process `listenForMessage` (Line 7). On successful termination the process `listenForMessage` passes the control to the succeeding behavioral block (`>>` operator). If a valid message has been received, indicated by the `hand_to_tu` variable, then the request is forwarded to the transaction user (Line 11) and the state machine moves to the *Trying*-state (Line 13).

If the state machine is in the *Trying*-state (Line 23), any response from the transaction user (Line 24) is sent to the environment of the Registrar (Line 25). After that the state machine moves on to the *Completed*-state (Line 27).

In the *Completed*-state there is a non-deterministic internal choice between receiving a retransmission (Lines 32-36) and finally moving to the *Terminated*-state (Line 42). This non-deterministic choice is an abstraction for the timeout transition in the state machine.

Note that this non-deterministic choice is not the only source of non-determinism within our specification. Another non-deterministic choice within the SIP Registrar specification deals with user authentication. This is a feature that is not necessarily

```
process serverTransaction [pin, pout, from_tu, to_tu](
   trans_state: TransState,
   branch: Branch,
   response: SipResp) : noexit :=

   [trans_state eq ts_terminated] -> (
      listenForMessage[pin,pout,from_tu,to_tu](branch,response) >>
      accept msg: RegisterMsg, resp: SipResp, hand_to_tu: Bool in
      (
         [hand_to_tu] -> (
            to_tu !msg;
            serverTransaction[pin, pout, from_tu, to_tu](
               ts_trying, getBranch(msg) + 1, resp)
         )
         []
         [not(hand_to_tu)] -> (
            serverTransaction[pin, pout, from_tu, to_tu](
               ts_terminated, 0, resp)
         )
      )
   )
   []
   [trans_state eq ts_trying] -> (
      from_tu ?resp: SipResp;
      pout !resp;
      serverTransaction[pin, pout, from_tu, to_tu](
         ts_completed, branch, resp)
   )
   []
   [trans_state eq ts_completed] -> (
      (
         i;
         pin ?msg: RegisterMsg [(getBranch(msg) eq branch)];
         pout !response;
         serverTransaction[pin, pout, from_tu, to_tu](
            ts_completed, branch + 1, response)
      )
      []
      (
         i;
         serverTransaction[pin, pout, from_tu, to_tu](
            ts_terminated, 0, response)
      )
   )
endproc
```

Fig. 9. LOTOS specification of the transaction handling state machine of a Session Initiation Protocol Registrar.

turned on in a SIP Registrar. As we do not want to have two similar specification that just differ in the authentication handshake we have a non-deterministic choice between the authenticated and the unauthenticated mode of a SIP Registrar.

According to the RFC [RSC$^+$02] a SIP Registrar may reject a message with a short expiration interval or it may accept this message. This requirement again introduces some non-determinism in our SIP Registrar model.

3.2 Simplifications

As one can already see from the LOTOS example above, modeling always includes choosing proper simplifications. Although modern specification languages have high expressive power, it is impractical and often infeasible to model the complete concrete behavior of a system. Thus, when developing a formal model one usually abstracts from the real world.

Basically we distinguish between two different types of simplifications: *abstractions* and *limitations*. Abstractions are simplifications that preserve conformance. For example, one may only specify the behavior for a particular set of inputs; for the unspecified inputs the systems may behave arbitrarily. Hence, in the context of **ioco** with partial models, abstraction may constrain the inputs and may remove constraints from the output behavior. In contrast, a limitation is a restriction of the system's possible reactive behavior (output) and hence not a proper abstraction. Consequently, limitations do not preserve conformance and the tester must be careful in interpreting a fail verdict: it might be due to a limitation in the model.

In model-based testing the simplifications influence the kind of detectable faults. The more abstract or limited a formal model is, the less information for judging on the correctness of an implementation is available [Gau95]. Thus, a major challenge in deriving models for industrial applications is the selection of proper simplifications. Simplifications need to limit a specification's state space to a manageable size, while the model still needs to be concrete enough to be useful.

According to [PP05, PPW$^+$05] we distinguish five classes of abstractions: *functional*, *data*, *communication*, *temporal*, and *structural* abstractions. *Functional* abstraction focuses on the functional part of the specification. This class of abstractions comprises the omission of behavior that is not required by the objectives of the model. *Data* abstraction subsumes the mapping from concrete to abstract values. Data abstraction includes the elimination of data values that are not needed within the functional part of the specification. *Communication* abstraction maps complex interactions to a more abstract level, e.g., the formal model uses one message to abstract a handshake scenario (several messages) of the real world. *Temporal* abstraction deals with the reduction of timing dependencies within the formal specification. For example, a certain model specifies only the ordering of events, but abstracts from discrete time values. *Structural* abstraction combines different real world aspects into logical units within the model.

Simplifications for the SIP Registrar Model When developing the formal model of a SIP Registrar we have chosen the simplifications listed in Table 2.

In particular, we simplify our model with respect to general server errors (Simplification 1), because of the loose informal specification of server errors within the RFC. Server errors may occur at any time when the Registrar encounters an internal error. For testing general server errors we would need a significant knowledge about the implementation internals. Especially, in order to trigger a server error during test execution, we would need to know how to enforce it. Hence, we skip server errors from the modeled behavior which may result in a wrong testing verdict. Therefore, this simplification is a limitation.

Simplification 2 omits specification details about forwarding requests. Thus, we

Table 2. Simplifications for the specification of the SIP Registrar.

id	type	description
1	limitation	Our formal model of the Registrar never terminates with a server error.
2	limitation	Our specification never forwards REGISTER messages to other SIP Registrars.
3	limitation	We assume that the communication channel is reliable and delivers messages in the sent order.
4	functional	The Registrar starts from a well known initial state.
5	functional	While the authentication handshake is in our model, the calculation of authentication credentials is not modeled.
6	functional	REGISTER messages do not contain any REQUIRES header fields.
7	data	The CALL-ID is abstracted to the range $[0, 1]$.
8	data	We limit the integer part of the CSEQ header to $[0, 1]$. The method part is not in the formal model.
9	data	The range $[0, 2^{32-1}]$ of the EXPIRES header field can be divided into three partitions where we use only boundary values of each partition.
10	data	Our model uses three different users: An authorized user, a known but unauthorized user and an unknown user.
11	data	Our formal model uses three different CONTACT values: *, any_addr1, and any_addr2.
12	data	The TO and FROM header fields are omitted in our abstract REGISTER messages.
13	temporal	Our specification does not use any timers. We only focus on the ordering of events.

do not generate tests for this feature. This is again a limitation as the forwarding of requests would result in different outputs, i.e. the receiver of the forwarded request responds to the REGISTER message.

Simplification 3 removes the needs for modeling possible interleaving of messages. During test execution this assumption is ensured by running the test execution framework and the implementation under test on the same computer.

Simplification 4 requires the start from a defined initial state. Otherwise, our model would have to consider different database contents on startup of the Registrar. We consider this a functional abstraction, because the functionality for other initializations is left open.

Simplifying the model with respect to the calculation of authentication credentials (Simplification 5) does not impose any limitation if the credentials are calculated and inserted correctly into test messages during test execution. As the detailed algorithm for credentials calculation is abstracted this is a functional abstraction.

We also skipped the REQUIRES header field in the formal specification in order to limit the number of possible request messages (Simplification 6). Not considering this input header field as being part of a REGISTER request represents a functional abstraction: the implementation may behave arbitrarily after this unspecified request.

Simplifications 7-10 are based on the ideas of equivalence partitioning and boundary value analysis [Mye79], which are strategies from white-box testing. For example, Simplification 10 uses the fact, that the Registrar relevant part of the RFC only distinguishes users that (1) are known by the proxy and allowed to modify contact information, (2) that are known by the proxy but are not allowed to modify contact information, and (3) users that are not known by the proxy, i.e. users that do not have an account. Thus, only three different users are needed, one of each group. Note that the simplifications 7-10 are data abstractions. Each of the header fields addressed by these abstractions are inputs to our specification. However, as we have one value per equivalence partition this is not a functional abstraction: every behavior (with respect to these inputs) of the system is modeled. Nevertheless, as we do not model all possible input values this is a data abstraction.

Simplification 11 limits the different CONTACT header field values. We allow the two addresses “any_addr1” and “any_addr2”, respectively. These two elements are replaced during test execution with valid contact addresses. According to the RFC, the asterisk is used for “delete” requests. This is again a data abstraction as the different possible behaviors are covered by our specification.

Simplification 2 causes the header fields, TO and FROM, to contain redundant information. So they can be omitted from our formal REGISTER messages (Simplification 12). Again this is a data abstraction.

Finally, as TGV does not support real-time testing, we need to abstract from concrete time events (Simplification 13).

4 Fault-based Conformance Testing

Given a formal specification there is a huge, possibly infinite, number of test cases that can be derived from that specification. There are different ways of selecting a finite set of test cases. One possibility for test case selection is the use of coverage criteria (e.g. [CR93, FWW08a]) on the level of the specification. Another way is the use of anticipated faults for the generation of test cases. This idea dates back to the late 1970s [DLS78, Ham77] where testers mutated source code to assess their test cases. Budd and Gopal [BG85] applied this technique to specifications.

In this paper, we also consider specification mutation as a way to select test cases. A fault is modeled at the specification level by altering the specification syntactically. The idea is to generate test cases that would fail if an implementation conforms to a faulty specification [AD06]. In order to generate test cases we mutate LOTOS specifications. Every mutant is compared to the original specification with respect to input-output conformance. If the mutant does not conform to the specification, then the trace leading to non-conformance serves as a test purpose. This test purpose is fed into the TGV tool [JJ05] in order to derive tree structured test cases which can be applied to non-deterministic systems.

Thus, we generate a test purpose for a specification S as follows:

1. Select a mutation operator O_m.
2. Generate a mutated version S^m of the specification S by applying O_m to S.

3. Check S and S^m for input output conformance (using an ioco checker [WW08a]).
4. Use the counterexample c, if any, as a test purpose TP[1] for the TGV tool.
5. Run the TGV tool with the test purpose TP on the original specification S in order to derive the final test case.

As we introduce faults at the level of the specification mutation operators are needed. These operators represent the sort of faults that we consider. For the selection of mutation operators one usually relies on two hypotheses [BDLS80]. The first one is called the 'competent specifier hypothesis' which is related to the 'competent programmer hypothesis' [BDLS80]. This hypothesis states that the specifier (programmer) is usually competent and gets the specification (program) almost correct. Faults can be corrected by a few key strokes.

The second assumption is called the 'coupling hypothesis'. It states, that big and dramatic effects that arise from bugs in software are closely coupled to small and simple failures.

Due to these two assumptions we can stick to small mutations on the specification. Thus, as usual in mutation testing, we use small syntactic changes on LOTOS specifications; each mutant only comprises a single mutation.

We use some of the mutation operators proposed in [BOY00, SCSP03] and adapted them to LOTOS specifications. Our mutation operators listed in Table 3.

4.1 Fault-based IOCO Testing

We are interested in testing for input-output conformance. Thus, for every mutant we want to generate a test case, such that the test case fails if this mutant has been implemented. However, not all mutation operators lead to models that can be distinguished from the original specification when using **ioco**, i.e. not all mutations represent faults. A fault can only be defined with respect to a conformance relation.

A mutant that cannot be distinguished from the original specification is called equivalent mutant. Although, the **ioco** relation is not an equivalence relation, we still use the terms equivalent and non-equivalent mutant as they are common in mutation testing. For an equivalent mutant there is no test case that distinguishes the mutant from the original specification. Contrary, a non-equivalent mutant comprises a fault such that there is a test case that passes on the original specification and fails on the mutant.

In the following the meaning of faults in the context of **ioco** is shown. Our first observation is that not all injected faults will cause observable failures. In order to observe a failure, the mutant must not conform (with respect to **ioco**) to our original specification. Hence, given an original specification S we are only interested in mutants S^m, such that[2]

$$\neg\ (S^m\ \mathbf{ioco}\ S)$$

Unfolding the definition of **ioco** gives

$$\neg\ (\forall \sigma \in \mathit{Straces}(S) \bullet \mathit{out}(S^m\ \mathbf{after}\ \sigma) \subseteq \mathit{out}(S\ \mathbf{after}\ \sigma)) \tag{1}$$

[1] The labels of the test processes are marked with *INPUT* or *OUTPUT*. We remove these marks. Furthermore, we have to add Refuse and Accept states.

[2] We make S^m input enabled.

Table 3. Mutation operators for LOTOS specifications.

Op.	Name	Description
ASO	Association Shift Op.	Change the association between variables in boolean expressions, e.g. replace $x \wedge (y \vee z)$ by $(x \wedge y) \vee z$.
CRO	Channel Replacement Op.	Replace the communication channel, i.e. change the gate of an event. For example, this operator would change Line 34 of Figure 9 from `pout !response` to `pin !response`.
EDO	Event Drop Op.	Drop events of the specification.
EIO	Event Insert Op.	Duplicate existing events.
ENO	Expression Negation Op.	Replace an expression by its negation, e.g. replace $x \wedge y$ by $\neg(x \wedge y)$.
ERO	Event Replacement Op.	Replace an event by a different event. For example, applying this operator to Line 34 of Figure 9 leads to `pout !response` to `pout`.
ESO	Event Swap Op.	Swap two neighbouring events.
HDO	Hiding Delete Op.	Delete an event from hide definition, i.e. make an unobservable event observable.
LRO	Logical Operator Replacement	Replace a logical operator by other logical operators.
MCO	Missing Condition Op.	Delete conditions from decisions.
ORO	Operand Replacement Op.	Replace an operand (variable or constant) by another syntactically legal operand, e.g. on mutation with respect to this operator is to replace the branch variable in Line 33 of Figure 9 by 0.
POR	Process Operator Replacement	Replace synchronization operators (||,|[...]|,|||)
PRO	Process Replacement Op.	Replace process instantiations with stop or exit events
RRO	Relational Operator Replacement	Replace a relational operator ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$) by any other except its opposite (since the opposite is similar to the negation operator)
SNO	Simple Expression Negation	Replace a simple expression by its negation, e.g. negate x in $x \wedge y$ getting $(\neg x \wedge y)$.
SOR	Sequential Operator Replacement	Replace the sequential composition operator >> and [>.
USO	Unobservable Sequence Op.	Make events of the specification unobservable.

which can further be rewritten to

$$= \exists \sigma \in Straces(S) \bullet out(S^m \textbf{ after } \sigma) \not\subseteq out(S \textbf{ after } \sigma) \tag{2}$$

This is the first hint for a testing strategy. We are interested in the suspension trace of actions leading to non-conformance between the mutant and the original LTS. In other words, our test purposes for detecting faults are sequences of actions leading to non-conformance. As one can see from this formula, a non-conformance check can be reduced to a test for subsets on the output labels.

It follows that a failure is observed, if the mutant S^m produces an output o not predicted by specification S:

$$= \exists \sigma \in Straces(S) \bullet \exists o \bullet o \in out(S^m \textbf{ after } \sigma) \wedge o \notin out(S \textbf{ after } \sigma) \qquad (3)$$

This simple derivation shows an important property of the **ioco** relation: mutating the specification by injecting an additional input a such that a new trace for the mutant S^m is generated, i.e. $\forall \sigma \in Straces(S) \bullet \sigma \cdot a \notin Straces(S)$, does not lead to a failure.

The theory highlights a further important clarification in fault-based testing: In the presence of non-determinism, there is no guarantee that an actual fault will always be detected. The reason is that non-conformance only means that there is a wrong output after a trace of actions, but the implementation may still opt for the correct one. In that case we rely on the complete testing assumption [LvBP94], which says that an implementation exercises all possible execution paths of a test case t, when t is applied a predetermined finite number of times.

4.2 On-the-fly IOCO Checking

As highlighted above we only need to consider mutants S^m of a specification S such that $\neg(S^m \textbf{ ioco } S)$. Because the state spaces of specifications are usually huge, we cannot construct the state space of the specification and the mutant in advance, and then check for conformance. Thus, conformance checking between the mutant and the specification has to be done on the fly.

Therefore, we use the LOTOS parser of the CADP toolbox [GLM02]. This parser takes a LOTOS specification and allows one to access the underlying LTS incrementally. The LOTOS specification is translated into an initial-state and a successor-function. The successor-function takes a state and returns the edges, i.e. labels and end-states, that are enabled in the given state.

In order to check two labeled transition systems for conformance we use an approach similar to the approach of Fernandez and Mounier [FM91]. That is, we define a synchronous product ($\times_{ioco}$) between two labeled transition systems S^m and S such that $S^m \times_{ioco} S$ contains special fail states if $\neg(S^m \textbf{ ioco } S)$. Checking for conformance is then implemented as a simple reachability search for fail states. If there is a fail state after a particular path, then this path is a counter-example showing the non-conformance between S^m and S.

Since, the input output conformance relation uses δ-labeled transitions and these transitions are not initially provided by the semantics of LOTOS specification we have to identify and to label quiescent states before calculating the synchronous product. More precisely, we add quiescence labeled transitions for quiescent states when iterating over the transitions of a particular state.

After adding the quiescence information we make the two labeled transition systems deterministic. This is done during the calculation of the synchronous product by the use of the subset construction [HU79]. Note that in the worst case this may cause an exponential increase of the number of states. During the process of making the LTSs deterministic we remove τ-labeled transitions too.

Definition 14. Let $S^m = (Q^{S^m}, L \cup \{\tau, \delta\}, \rightarrow_{S^m}, q_0^{S^m})$ and $S = (Q^S, L \cup \{\tau, \delta\}, \rightarrow_S, q_0^S)$ be two deterministic LTSs, where the labels L are partitioned into inputs L_I

and outputs L_U, i.e. $L = L_I \cup L_U$ and $L_I \cap L_U = \emptyset$. The synchronous product $SP = S^m \times_{ioco} S$ is an LTS $SP = (Q^{SP}, L, \rightarrow_{SP}, q_0^{SP})$, where its state set Q^{SP} is a subset of $(Q^{S^m} \times Q^S) \cup \{pass, fail\}$ reachable from the initial state $q_0^{SP} =_{df} \left(q_0^{S^m}, q_0^S\right)$ by the transition relation $\rightarrow_{SP}$. Let $q^{S^m} \in Q^{S^m}$ and $q^S \in Q^S$ be two states of S^m and S, respectively. Then, the transition relation $\rightarrow_{SP}$ is defined as the smallest set obtained by the application of the following rules:

1. Edges possible in both LTSs, S^m and S:
   $\forall a \in L_I \cup L_U \bullet \forall q'^{S^m} \in Q^{S^m}, q'^S \in Q^S \bullet q^{S^m} \xrightarrow{a}_{S^m} q'^{S^m} \wedge q^S \xrightarrow{a}_S q'^S \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{a}_{SP} \left(q'^{S^m}, q'^S\right)$.
2. Implementation freedom on unspecified inputs:
   $\forall a \in L_I \bullet \forall q'^{S^m} \in Q^{S^m} \bullet q^{S^m} \xrightarrow{a}_{S^m} q'^{S^m} \wedge q^S \overset{a}{\not\rightarrow}_S \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{a}_{SP} pass.$
3. S^m may allow fewer outputs than S:
   $\forall b \in L_U \bullet \forall q'^S \in Q^S \bullet q^S \xrightarrow{b}_S q'^S \wedge q^{S^m} \overset{b}{\not\rightarrow}_{S^m} \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{b}_{SP} pass.$
4. Input enabledness of S^m:
   $\forall a \in L_I \bullet \forall q'^S \in Q^S \bullet q^S \xrightarrow{a}_S q'^S \wedge q^{S^m} \overset{a}{\not\rightarrow}_{S^m} \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{a}_{SP} \left(q^{S^m}, q'^S\right)$.
5. Unspecified outputs of S^m:
   $\forall b \in L_U \bullet \forall q'^{S^m} \in Q^{S^m} \bullet q^{S^m} \xrightarrow{b}_{S^m} q'^{S^m} \wedge q^S \overset{b}{\not\rightarrow}_S \Rightarrow \left(q^{S^m}, q^S\right) \xrightarrow{b}_{SP} fail.$

□

Rule 1 states, that edges that are possible in both LTSs are edges of the synchronous product.

Rule 2 handles the cases where the LTS representing the implementation allows inputs, that are not specified by the LTS representing the specification. Since **ioco** allows any behavior on unspecified inputs we add a pass state to the synchronous product. The added state is a sink state, i.e., there are no outgoing edges. Note, that pass states do no affect the final comparison result, since only the existence of fail states determine whether two LTS are related under conformance (with respect to **ioco**) or not.

Since, **ioco** requires that the outputs of the implementation's LTS have to be a subset or have to be equal to the outputs of the specification's LTS we add an edge to a pass state for any output that is allowed in S but not in S^m (Rule 3). Again, this pass state has no influence on the final comparison result.

Note, that **ioco** requires the left hand side LTS to be weakly input enabled. In practice this may not be the case for a given input output labeled transition system. Thus, we have to convert the left hand side LTS to a weakly input enabled LTS. The synchronous product considers this requirement by Rule 4 of the transition relation. This rule assumes that an input is always possible in S^m. Thus, if input a is not allowed in S^m, the input enabledness allows to assume a self-loop labeled with a on state q^{S^m}. Hence, this rule ensures that the synchronous product will not contain an edge leading to fail because S^m lacks input enabledness.

We add an edge leading to a fail state if an output of the left hand LTS S^m is not an output of the right hand LTS S (Rule 5). Only in that case the two LTSs do not conform with respect to **ioco**.

Example 9. Figure 11 shows the synchronous products used to check conformance

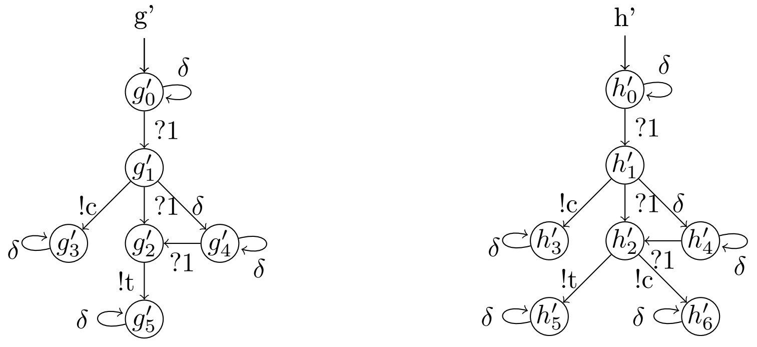
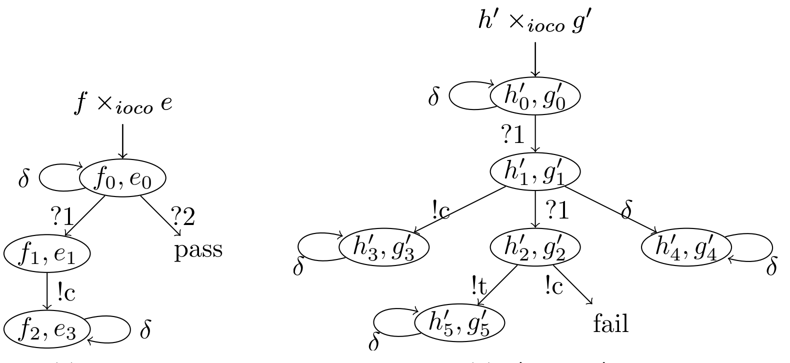


Fig. 10. Deterministic versions of the labeled transition systems g and h of Figure 3.



(a) $f \times_{ioco} e$ (b) $h' \times_{ioco} g'$

Fig. 11. Synchronous products illustrating the application of Definition 14 to the labeled transition systems f and e of Figure 3 and g' and h' of Figure 10.

on some of the labeled transition systems of Figure 3. The basis of the **ioco** check are δ-annotated deterministic labeled transition systems.

When checking f **ioco** e the synchronous product according to Definition 14 looks like the LTS shown in Figure 11a. The trace $\langle ?2 \rangle$ of the implementation's model f is not relevant in specification e, and hence Rule 2 of Definition 14 applies.

When checking g **ioco** h and h **ioco** g the two labeled transition systems are turned deterministic first. The resulting LTSs g' and h' are illustrated in Figure 10. The synchronous product using $\times_{ioco}$ is illustrated in Figure 11b.

The fail state comes from Rule 5 which says, that outputs of the implementation that are not allowed by the specification lead to fail. However, when checking the reverse relation g **ioco** h there would be a pass state instead of the fail state. This is because implementations may have fewer outputs than specifications after a particular trace (Rule 3).

As the synchronous products are constructed on-the-fly the construction stops if a fail-state is reached. □

4.3 Handling large state spaces

IOCO checking of two conforming labeled transition systems requires the comparison of their whole state spaces. For large industrial specifications this is often infeasible. In the case of our SIP Registrar specification, the attempt of constructing the specification's state space on a computer with 2GB RAM failed after 11 days due to insufficient memory. Note that for this experiment we bound all used data types of our specification.

To overcome this problem, we proposed to exploit the knowledge where the fault has been inserted in the LOTOS specification [APWW07a]. By marking the place of the mutation it is possible to construct the relevant part of the state space only, i.e. the part that reflects the mutation. However, since the position of the markers is determined syntactically, our approach is not applicable for all mutation operators. For example, for the SIP Registrar specification, we succeeded to mark 252 out of 843 mutants only, i.e. our approach applied to only 30% of the mutants [AWW08].

Nevertheless, the on-the-fly approach for checking the conformance of two specifications is suitable for any mutation operator. We can apply the conformance check not to the whole specification's state space but only up to a particular depth. Thus, similar to bounded model checking [BCC$^+$03], we check for bounded input-output conformance:

Definition 15 (Bounded input-output conformance). Given a set of inputs L_I and a set of outputs L_U then $\mathbf{ioco}^{|k|} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I, L_U)$ is defined as:

$$IUT\ \mathbf{ioco}^{|k|}\ S =_{df} \forall \sigma \in Straces(S) \bullet$$
$$(length(\sigma) \leq k) \implies (out(IUT\ \mathbf{after}\ \sigma) \subseteq out(S\ \mathbf{after}\ \sigma))$$
□

Example 10. For example, let the input output transition system k of Figure 4 be the specification and let the IOTS l of Figure 4 be the model of an implementation. l does not (**ioco**-) conform to k, i.e. $\neg(l\ \mathbf{ioco}\ k)$, because $out(l\ \mathbf{after}\ \langle ?1, \delta, ?1 \rangle) = \{!c, !t\} \not\subseteq \{!t\} = out(k\ \mathbf{after}\ \langle ?1, \delta, ?1 \rangle)$. However, we have $l\ \mathbf{ioco}^{|0|}\ k$, because $out(l\ \mathbf{after}\ \langle\rangle) = \{\delta\} = out(k\ \mathbf{after}\ \langle\rangle)$. Furthermore, we have $l\ \mathbf{ioco}^{|1|}\ k$, because $l\ \mathbf{ioco}^{|0|}\ k$, $out(l\ \mathbf{after}\ \langle\delta\rangle) = \{\delta\} = out(k\ \mathbf{after}\ \langle\delta\rangle)$, $out(l\ \mathbf{after}\ \langle ?1\rangle) = \{!c, \delta\} = out(k\ \mathbf{after}\ \langle ?1\rangle)$, and $out(l\ \mathbf{after}\ \langle ?2\rangle) = \{\delta\} = out(k\ \mathbf{after}\ \langle ?2\rangle)$. We also have $l\ \mathbf{ioco}^{|2|}\ k$, because $l\ \mathbf{ioco}^{|0|}\ k$ and $l\ \mathbf{ioco}^{|1|}\ k$ and there is no trace after which an output of l is not allowed by k. The shortest trace leading to non-conformance has a length of three, i.e. $\langle ?1, \delta, ?1\rangle$. Thus, with a bound greater or equal to three l does not conform to k, i.e. $\neg(l\ \mathbf{ioco}^{|3|}\ k)$. □

In contrast to our previous syntactic labelling technique, bounded input-output conformance checking applies to any mutation operator. If we find a counter-example when checking for $S^m\ \mathbf{ioco}^{|k|}\ S$ within a particular bound k, then the counter-example is also valid for showing non-conformance for **ioco**. However, failing to show non-conformance within a particular bound k does not mean that we necessarily have an ioco-correct, i.e. an equivalent, mutant. Thus, the technique is sound but incomplete. Here, soundness guarantees that no counterexamples for equivalent mutants are generated. Hence, we will never produce redundant test cases from equivalent mutants. Due to incompleteness we may miss some test cases aiming for faults that are observable only above the boundary.

Table 4. Details of the LOTOS specifications of the applications under test.

	No.processes	No.actions	No.datatypes	net. Lines of Code	
				Total	data types
SIP	10	27	20	3000	2500
CP	16	26	1	900	700

5 Experimental Results

This section presents the results obtained when applying our approach to two different applications: (1) The Session Initiation Protocol (SIP) [RSC+02] and (2) the Conference Protocol (CP) [TPHT96]. The results are presented in terms of source code coverage, and actual faults found in comparison to testing using manual test case selection and random test case selection. We conducted all our experiments on a PC with Intel(R) Dual Core Processor 1.83GHz and 2GB RAM.

5.1 Applications under Test

In addition to the LOTOS specification of a SIP Registrar, which comprises approximately 3KLOC (net.), 20 data types (contributing to net. 2.5KLOC), and 10 processes, the Conference Protocol serves to evaluate our approach.

Table 4 summarizes the characteristics of the two specifications in terms of number of processes, number of actions, and in terms of net lines of code.

5.1.1 The Conference Protocol

The Conference Protocol has been used previously to analyze the fault-detection ability of different formal testing approaches (e.g., [dBRS+00, BFdV+99]). The specification is available in different specification languages to the public[3]. In addition, there are 27 erroneous implementations which can be used to evaluate testing techniques.

The protocol itself is a simple communication protocol for a chat application. The main part of the application is called the Conference Protocol Entity (CPE). A CPE serves as chat client with two interfaces; one interface allows the user to enter commands and to receive messages sent by other users, and the other interface allows the chat application to send and receive messages via the network layer. These two interfaces are the points of control and observation for a tester.

Users can join conferences, exchange messages and leave conferences. Each user has a nickname and can join one conference at a time only. Figure 12 shows a typical example of a simple chat session. First, a user with nickname Bob joins conference “C1”. The Conference Protocol Entity sends that information to all potential conference partners. In the illustrated scenario user Alice participates in the same conference as joined by Bob. Thus, Alice’s protocol entity answers with an *answer*-protocol data unit (PDU). Then Alice decides to leave the conference which causes her protocol entity to send a *leave*-PDU to all participating users, i.e., to Bob’s CPE.

[3] `http://fmt.cs.utwente.nl/ConfCase/`

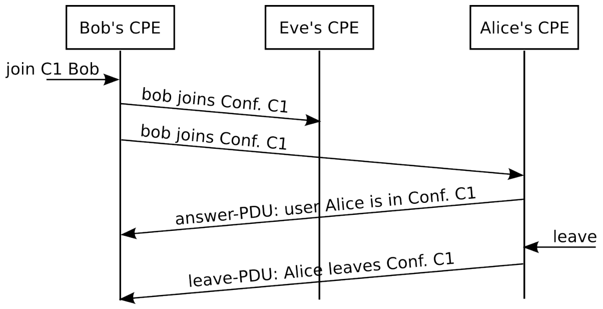


Fig. 12. Simple call-flow illustrating joining and leaving a chat session.

5.2 Test Case Generation Results

We developed a mutation tool that takes a LOTOS specification and uses the mutation operators of Section 4 in order to generate for each possible mutation one faulty version (mutant) of the specification. With this tool all mutants of the specifications were generated automatically.

Table 5 lists for each mutation operator (1st column) the overall number of generated mutants (2nd and 7th column) for the session initiation protocol specification and for the Conference Protocol specification. The 3rd (8th) column depicts the average time needed for running the on-the-fly input-output conformance check.

We have set the bound of the depth first search for the SIP Registrar to 5 steps and for the Conference Protocol to 10 steps. Note that internal transitions do not add to the length of a trace, i.e. a bound of five means that we checked for conformance using all traces comprising five or less visible actions.

The number of equivalent and different mutants (with respect to the **ioco** relation) are listed in the 4th and the 5th column of Table 5 for the SIP Registrar specification and in the 9th and the 10th column for the Conference Protocol specification. Finally, the 6th and the 11th columns list the average time needed by the TGV tool to extract a final test case.

Note that the third, the sixth, the eighth and the eleventh columns of Table 5 list the average duration needed for the different steps during the test case generation. Thus, also the last row (Σ) lists the average values in these columns.

Approximately, 47% (31%) of the generated mutants for the SIP Registrar (Conference Protocol) specification are distinguishable from the original specification with respect to $\mathbf{ioco}^{|k|}$. The other mutants do not result in useful test cases when testing for particular faults. Although, the chosen bound of the SIP Registrar is smaller than the bound used for the Conference Protocol, the conformance check on the SIP specification was slower. This is because the SIP Registrar branches heavily in the beginning, i.e. there are approximately 2700 outgoing transitions at the initial state.

Remarkably the two mutation operators ASO and SOR did not lead to any mutant with an observable difference for the two specifications. Within the Conference Protocol there are no logical expressions comprising more than one logical operator.

[4] Note that the difference between this number and the 843 mutants reported in [AWW08] comes from minor structural improvements of our specification.

Table 5. Number of generated mutants and timing results for the extraction of the test cases for the two considered protocol specifications.

oper-	SIP Registrar					Conference Protocol				
ator	mut.	ioco	$=$	$\neq$	tgv	mut.	ioco	$=$	$\neq$	tgv
ASO	11	4m52s	11	0	-	0	-	0	0	-
CRO	81	4m58s	26	55	5s	20	11s	12	8	1s
EDO	31	5m09s	14	17	5s	10	14s	7	3	1s
EIO	35	7m41s	24	11	5s	12	15s	9	3	1s
ENO	52	4m49s	16	36	5s	38	08s	21	17	1s
ERO	31	4m39s	7	24	5s	10	04s	4	6	1s
ESO	10	7m12s	8	2	5s	0	-	0	0	-
HDO	4	4m28s	1	3	5s	0	-	0	0	-
LRO	21	8m15s	13	8	5s	15	18s	15	0	-
MCO	33	8m12s	26	7	5s	30	18s	30	0	-
ORO	431	7m09s	245	186	5s	183	1m35s	126	57	1s
POR	6	32m39s	2	4	5s	3	07s	2	1	1s
PRO	18	5m03s	8	10	5s	30	09s	14	16	1s
RRO	91	7m42s	59	32	5s	0	-	0	0	-
SNO	32	6m57s	17	15	5s	39	18s	28	11	1s
SOR	0	-	0	0	-	0	-	0	0	-
USO	23	4m38s	5	18	5s	10	18s	8	2	1s
Σ	910[4]	6m26s	482	428	5s	400	23s	276	124	1s

Thus the ASO operator does not produce any mutant on that specification. Contrary, there are logical expressions within the SIP Registrar specification that use more than one logical operator. However, these expressions always use the same logical operator, thus an association shift within these expression does not lead to any observable difference.

As there are no enabling (`>>`) nor any disabling (`[>`) statements within the Conference Protocol specifications the SOR operator does not produce any mutant. The SIP Registrar specification comprises six `>>` statements. However, in this specification replacing `>>` with `[>` always resulted in an invalid specification. This is because for the `[>` operator the exit-behavior, i.e. the types of the returned values, has to be equal. Unfortunately, the exit-behavior always differs in our specification.

Anyway, on other specifications these mutation operators can lead to mutants exhibiting an **ioco** incorrect behavior.

5.3 Test Case Execution Results

The test cases were executed on real implementations of both protocols. As we rely on TGV for test case generation the obtained test cases are not input enabled, i.e. our test cases may require to prevent an implementation from doing outputs (see Definition 9).

However, in practice it is often not possible that a tester prevents an implementation from doing outputs. One way to overcome this issue is to apply the *reasonable* *environment assumption* [FJJV97], which says that before the environment sends a message to the network, it waits until stabilization. This means that the test execution environment is not allowed to send new messages until it received all responses from the implementation. In order words, one gives outputs (of the IUT) priority over inputs (to the IUT).

Recall, that TGV prunes edges during test case generation, as this tool relies on blocking outputs from the IUT while inputs are enabled. Thus, we implement the reasonable environment assumption by running the specification in parallel to the test case execution. If there is an input from the implementation to a test case and this input is allowed by the specification but not by the test case we give an inconclusive verdict. If the input is not allowed by both the test case and the specification we give a fail verdict. Otherwise, i.e. the input is allowed by the test case, we continue the execution of the test case.

Note that the reasonable environment assumption is not a general assumption of our approach. We only used it to ensure correct verdicts during test case execution. If the assumptions of **ioco** are satisfied then there is no need for using the reasonable environment assumption.

In a previous project, we tested a commercial implementation of the Session Initiation Protocol Registrar [APWW07a, AWW08]. Because this implementation is no longer available to us, in this paper the open source implementation Kamailio[5] serves as implementation under test.

Table 6 illustrates the number of passed (3rd and 7th column), failed (4th and 8th column) and inconclusive (5th and 9th column) verdicts obtained by executing the generated test cases. The number of test cases is listed in the 2nd column and the 6th column, respectively. Note, that we run the test cases of the SIP Registrar on two different configurations of the Kamailio implementation. For one test run authentication was turned on and for the other test run authentication was turned off. For example, we run the 15 test cases derived from the SNO mutations two times, resulting in 30 test runs. 24 out of these 30 test runs terminated with a pass verdict while 5 test runs reported a fail verdict. One of the 30 test runs led to an inconclusive verdict. A test case's verdict is inconclusive if the implementation chooses outputs different to the outputs required by the test case's preamble. That is, the chosen output is correct with respect to the specification, but the test case failed to bring the implementation to the required state.

For the Conference Protocol we have 27 faulty implementations. Thus, running for example the 8 test cases derived from the CRO mutations we get $8 \times 27 = 216$ test runs. 208 out of these 216 test runs terminated with a pass verdict, while 8 test runs ended with a fail verdict.

By the use of the generated test cases we detected 4 differences between the Kamailio Registrar and our specifications. However, a verdict fail does not imply that the corresponding mutant has been implemented. It also happens that there occurred a failure during the execution of the preamble of the test case. The preamble is the

[5] Note, that Kamailio was previously named OpenSER and can be found at `http://www.kamailio.org/`

Table 6. Test case execution results for the two protocol specifications.

oper-	SIP Registrar				Conference Protocol			
ator	no.tc.	pass	fail	inconc.	no.tc.	pass	fail	inconc.
ASO	0	-	-	-	0	-	-	-
CRO	55	85	25	0	8	208	8	0
EDO	17	27	7	0	3	78	3	0
EIO	11	18	4	0	3	78	3	0
ENO	36	60	12	0	17	430	29	0
ERO	24	37	11	0	6	156	6	0
ESO	2	2	2	0	0	-	-	-
HDO	3	5	1	0	0	-	-	-
LRO	8	15	1	0	0	-	-	-
MCO	7	13	1	0	0	-	-	-
ORO	186	302	70	0	57	1458	81	0
POR	4	7	1	0	1	26	1	0
PRO	10	17	3	0	16	408	24	0
RRO	32	56	8	0	0	-	-	-
SNO	15	24	5	1	11	282	15	0
SOR	0	-	-	-	0	-	-	-
USO	18	29	7	0	2	52	2	0
Σ	428	698	158	1	124	3176	172	0

sequence of messages that aims to bring the implementation to a certain state in which the difference between the mutant and the original specification can be observed.

For the Conference Protocol we detected in total 7 of the 27 faulty implementations. Recall that the bound of the test case length is ten actions in the case of the Conference Protocol. This limitation in the length of the test cases is mainly the reason why we did not detect more faulty implementations.

5.4 Comparing Fault-based Testing to Other Approaches

In order to evaluate the quality of the generated test cases using our fault-based technique, we compared results of our approach to results obtained when using hand-crafted test purposes (scenarios), i.e. the TGV tool [JJ05], and when using random testing, i.e. the TORX tool [TB03].

We identified five relevant scenarios from the textual specification of the SIP Registrar and ten interesting scenarios from the textual description of the Conference Protocol. TGV derives only one test case per test purpose. As we have shown in [FWW08b], deriving multiple test cases for a single test purpose increases the source code coverage and the number of detected faults on implementations for the resulting test suite. We compared the results of our fault-based approach with test suites obtained when using one test case per test purpose and with test suites comprising multiple test cases per test purpose. For deriving multiple test cases per test purpose we use transition coverage on the (final) synchronous product between each test purpose and the specification.

Table 7. Overview of the main results using random, scenario-based and fault-based test case generation techniques.

IUT	Technique	seq. length	test gen.	test cases	avg. coverage		no. faults
					F	C/D	
SIP	random	10.95	4s	100	73%	36%	5
	scenarios (1)	2.20	1s	10	64%	26%	2
	scenarios (many)	4.53	1s	6813	73%	34%	6
	fault-based	3.75	6m26s	428	70%	31%	4
Conf. Prot.	random	9.03	4s	100	70%	56%	19
	scenarios (1)	5.53	1s	5	75%	58%	17
	scenarios (many)	4.98	1s	408	77%	60%	23
	fault-based	7.23	23s	124	66%	51%	7

For random testing we ran the TORX tool 100 times on every implementation. For our comparison we conducted three times 100 test runs, i.e. we made 100 random test runs and repeated this experiment three times with different seeds for the random value generator. The results shown in this paper are the average values out of these three experiments. Thus, for the one SIP Registrar we get 100 test runs (i.e. the average of 3 times 100), while for the Conference Protocol we have 2700 test runs, i.e. 100 test runs for each of the 27 faulty implementations.

Table 7 summarizes the results when testing the two protocols using different test case selection strategies. This table shows for each of the three techniques (2nd column), the average length of the executed test sequences (3rd column). The next columns depict, the average time needed to generate a single test case (4th column) and the overall number of generated test cases (5th column). In addition, Table 7 shows the code coverage[6] in terms of function coverage (6th column), and condition/decision coverage (7th column). Finally, Table 7 illustrates the number of detected faults (8th column).

The code coverage shows some interesting properties of the generated test cases. First of all, random testing covers less functions than the test cases derived from our scenarios. This is because there are some complex scenarios which require a particular sequence of test messages in order to put the implementations into certain states. For example, if the Registrar is in such a state it uses additional functions for processing REGISTER requests. Unfortunately, the random test generation never selected this sequence from the formal specification.

The condition/decision (C/D) coverage achieved by random testing is higher than the C/D coverage from scenario-based testing. That means, that random testing has inspected the covered functions more thoroughly. In the case of the SIP Registrar the fault-based test cases cover more source code and find more faults than one test case per scenario. For the Conference Protocol this is not the case. This shows, that scenario based testing highly depends on the skills of the tester.

For the Conference Protocol test cases generated based on scenarios detected all

[6] For coverage measurements we use the Bullseye Coverage Tool: http://www.bullseye.com

faults that are detected by the other approaches. That is, combining all test cases of all our experiments we detected 23 of the 27 faulty implementations. Using many test cases for a single scenario revealed faults that have not been detected using random or fault-based testing. The faulty implementations found by the use of fault-based testing have also been revealed using the other test case select techniques.

In the case of the SIP Registrar only many test cases per scenario detect the faults revealed by fault-based testing. Fault-based testing complements random testing and scenario based testing when one test case per scenario is used.

Thus, in practice it is recommendable to combine different test case selection techniques, because for example fault-based testing revealed additional faults in the tested implementations.

6 Related Research

There are different approaches for test cases selection, e.g. random or coverage based test case selection. While random test case selection was the first test case selection strategy considered for **ioco** testing [Tre96], coverage based testing for LOTOS specifications has been explored by Amyot and Logrippo [AL00] and by Cheung and Ren [CR93]. Also the work of Huo and Petrenko [HP08] deals with coverage based test case selection. However, they consider transition and state coverage of labeled transition systems.

While we consider fault-based conformance testing, mutation based test case generation by the use of model-checkers has been proposed by Ammann et al. [ABM98]. While model-checker based mutation testing was limited to deterministic models recent research [BPG07, OBY02, FW07] allows the application of model-checkers to non-deterministic models.

The work of Okun et al. [OBY02] considers the generation of counterexamples for non-determinism within the specification. If there is a non-deterministic choice within a specification and a mutant all possible combinations of the non-deterministic choices in the mutant and the original explored. This may lead to a counterexample coming from a difference in the execution, not from a semantic difference. The approach of Okun et al. solves this problem for some models.

Boroday et al. [BPG07] propose to use module checking [KV96] to cope with non-determinism. However, the proposed approach produces again a linear counterexample. There is no statement about how to apply such test cases to systems that can choose between providing different outputs.

Testing such systems by the use of linear counterexamples has been investigated by Fraser and Wotawa [FW07]. The basic idea of their work is to give inconclusive verdicts in states where the model comprises non-deterministic choices. If test case execution terminates with an inconclusive verdict, they use the model to verify if the output of the implementation is allowed by the specification. If there is an invalid output test case execution is terminated with a fail verdict. Otherwise, the test case is extended using the information obtained from the test case execution, and the test case is re-executed on the implementation. This procedure is repeated until the test case execution terminates with a pass or fail verdict. Contrary, we do not need to extend our test cases during test case execution.

Petrenko and Yevtushenko [PY05] showed how to use partial, non-deterministic finite state machines (FSM) for mutation based test case generation. This work makes FSM based testing more amenable in industrial applications where specifications are rarely deterministic and complete. The difference to our approach is the used model. FSMs assume that a system cannot accept a next input before producing an output as a reaction to a previous input.

Another work that considers the combination of fault-based testing and test purposes is [PBG04]. The used models are extended finite state machines (EFSM). The authors consider one single fault-type where an implementation is in an wrong post-state after applying a particular test sequence. The test purpose is given in terms of configurations of the EFSM denoting suspicious implementation states. That is, the test purpose describes outputs that should be avoided. The authors also consider limiting the length of the test sequence. In contrast we consider mutations on the level of the specification. Furthermore, our approach automatically derives a test purpose. This test purpose is then used for test case generation.

While we considered mutating LOTOS specifications, Stocks applied fault-based testing to Z specifications [Sto93]. Mutation testing of Estelle specifications has been considered in [DSMFDS99].

The idea of generating test purposes instead of generating test cases directly has been subject to previous research [HLU03, dSM06]. The authors of [dSM06] present a modified model-checking algorithm that allows to transform properties, given in computational tree logic (CTL), to test purposes. Henniger et al. [HLU03] automatically generate test purposes by identifying significant behavior of a system. Each significant behavior is converted to a test purpose. However, both articles do not consider testing for specific faults.

Various testing techniques have been applied to SIP [WLS04, SNLD02]. While the applied techniques deal with security aspects and performance issues, to our best knowledge none of them focuses on protocol conformance testing.

Modeling SIP using SDL or UML has been subject to publication previously [CvB03, SRS01a, SRS01b]. In difference to our formalization, the presented models are based on the outdated RFC 2543 [HSSR99]. The presented formalizations are not tailored to any special purpose and deal with the session management part of SIP. Contrary, our model is based on the currently valid RFC 3261 and targets the user management part of SIP. Furthermore, the aim of our specification is protocol conformance testing.

The Conference Protocol example has been used by other authors to assess test case generation techniques [BFdV^{+}99, dBRS^{+}00, HFT00, BN07, HP08]. A comparison when applying different tools to the Conference Protocol is given by Belinfante et al. [BFS04]. The applied tools detected between 21 and 25 of the 27 faulty implementations. Note that the missing two implementations comprise faults that cannot be detected using the **ioco** relation. Also the approach of Huo and Petrenko [HP08] revealed all 25 faulty implementations.

7 Conclusion

In this paper we presented our insights gained when testing industrial applications by the use of formal testing techniques. Particularly, we discussed the modeling of an industrial application, i.e. the Session Initiation Protocol Registrar. As our project was conducted together with an industrial partner, the specification has been reviewed by field experts.

We presented the chosen simplifications making the large state space of our specifications manageable. Nevertheless, the model still comprises enough information for deriving useful test cases.

We then showed how one can use a fault-based testing technique in order to prevent a system from implementing particular faults. Faults are modeled on the level of specifications. Faults are injected into specifications automatically. Such faulty specifications are called mutants. Every mutant contains only one fault. A conformance check between the mutant and the original specification leads to a counterexample (if any). This counterexample is then used as a test objective in order to generate a tree-structured test case. Such test cases are suitable for testing non-deterministic systems.

In the worst case one has to consider the whole state space of the mutant and the specification for this conformance check. This is not feasible for industrial scale applications. We showed how this can be solved by bounding the search depth for such a counterexample. Furthermore, we discussed the effects of having such a bound.

By applying our approach to two different specifications we demonstrated its feasibility in practice. While our technique does not substitute conventional test case selection strategies, it complements them. The experimental results showed that fault-based testing revealed an additional fault, not detected by random testing or scenario-based testing.

Further Research We consider fault-based **ioco** testing based on LOTOS specifications. Recent research incorporates data and data-dependent control flow into the **ioco** testing theory [RdBJ00, FTW06]. Further research may combine fault-based conformance with symbolic conformance testing. In symbolic conformance testing one does not need to enumerate all data for constructing the labels of the LTS but can use data directly within the symbolic transition system. This may be beneficial for data dependent specifications such as the SIP Registrar.

Another direction of research would be a closer investigation of the mutation operators with respect to the conformance relation. Due to the properties of **ioco**, i.e. additional new inputs do not lead to observable failures (see Section 4.1), one may determine in advance that some mutants do not exhibit an observable difference with respect to the original specification. Thus, one may omit some mutations during the fault injection phase of our approach. However, for this one needs to determine if the inserted event results in an additional new input within the mutant's underlying LTS.

Acknowledgments The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT). This work is also partially funded by the EU FP7 project MOGENTES ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems.

We are most grateful to the anonymous reviewers for their valuable comments.

References

[ABM98] P.E. Ammann, P.E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, pages 46–54, 9-11 Dec. 1998.

[AD06] Bernhard K. Aichernig and Carlo Corrales Delgado. From faults via test purposes to test cases: On the fault-based testing of concurrent systems. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, volume 3922 of *LNCS*, pages 324–338. Springer, 2006.

[AL00] Daniel Amyot and Luigi Logrippo. Structural coverage for lotos - a probe insertion technique. In *Proceedings of the 13th International Conference on Testing Communicating Systems: Tools and Techniques*, pages 19–34. Kluwer, B.V., 2000.

[APWW07a] Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Protocol conformance testing a SIP registrar: An industrial application of formal methods. In Mike Hinchey and Tiziana Margaria, editors, *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods*, pages 215–224, London, UK, 2007. IEEE.

[APWW07b] Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Test purpose generation in an industrial application. In *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, pages 115–125, London, UK, July 2007.

[AWW08] Bernhard K. Aichernig, Martin Weiglhofer, and Franz Wotawa. Improving fault-based conformance testing. *Electronic Notes in Theoretical Computer Science*, 220(1):63–77, December 2008.

[BCC$^+$03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[BDLS80] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233, New York, NY, USA, 1980. ACM.

[Ber91] Gilles Bernot. Testing against formal specifications: A theoretical view. In Samson Abramsky and T. S. E. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer, 1991.

[BFdV$^+$99] Axel Belinfante, Jan Feenstra, René G. de Vries, Jan Tretmans, Nicolae Goga, Loe M. G. Feijs, Sjouke Mauw, and Lex Heerink. Formal test automation: A simple experiment. In *12th International Workshop on Testing Communicating Systems*, volume 147 of *IFIP Conference Proceedings*, pages 179–196. Kluwer, 1999.

[BFS04] Axel Belinfante, Lars Frantzen, and Christian Schallhart. Tools for test case generation. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 391–438. Springer, 2004.

[BG85] Timothy A. Budd and Ajet S. Gopal. Program testing by specification mutation. *Computer languages*, 10(1):63–73, 1985.

[BN07] M. Botinčan and V. Novaković. Model-based testing of the conference protocol with spec explorer. In *Proceedings of the 9th International Conference on Telecommunications*, pages 131–138. IEEE, June 2007.

[BOY00] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 81–88, Grenoble, France, September 2000. IEEE.

[BPG07] Sergiy Boroday, Alexandre Petrenko, and Roland Groz. Can a model checker generate tests for non-deterministic systems? *Electr. Notes Theor. Comput. Sci.*, 190(2):3–19, 2007.

[CR93] T.Y. Cheung and S. Ren. Executable test sequences with operational coverage for lotos specifications. In *12th Annual International Phoenix Conference on Computers and Communications*, pages 245–253, 23-26 March 1993.

[CvB03] Ken Y. Chan and Gregor v. Bochmann. Modeling IETF session initiation protocol and its services in SDL. In *Proceedings of the 11th International SDL Forum: System Design*, volume 2708 of *LNCS*, pages 352–373. Springer, 2003.

[dBRS+00] Lydie du Bousquet, Solofo Ramangalahy, Séverine Simon, César Viho, Axel Belinfante, and René G. de Vries. Formal test automation: The conference protocol with TGV/TORX. In *Proceedings of 13th International Conference on Testing Communicating Systems: Tools and Techniques*, volume 176 of *IFIP Conference Proceedings*, pages 221–228, Dordrecht, August 2000. Kluwer Academic Publishers.

[DLS78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[dSM06] Daniel Aguiar da Silva and Patrícia D. L. Machado. Towards test purpose generation from CTL properties for reactive systems. *Electr. Notes Theor. Comput. Sci.*, 164(4):29–40, 2006.

[DSMFDS99] Simone Do Rocio Senger De Souza, José Carlos Maldonado, Sandra Camargo Pinto Ferraz Fabbri, and Wanderley Lopes De Souza. Mutation testing applied to estelle specifications. *Software Quality Control*, 8(4):285–301, 1999.

[EFH83] Hartmut Ehrig, Werner Fey, and Horst Hansen. Act one - an algebraic specification language with two levels of semantics. In Manfred Broy and Martin Wirsing, editors, *Proceedings 2nd Workshop on Abstract Data Type*, 1983.

[FJJV97] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146, 1997.

[FM91] Jean-Claude Fernandez and Laurent Mounier. “On the fly“ verification of behavioural equivalences and preorders. In *Proceedings of the 3rd International Workshop on Computer Aided Verification*, volume 575 of *LNCS*, pages 181–191. Springer, 1991.

[FTW06] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A symbolic framework for model-based testing. In *1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *LNCS*, pages 40–54. Springer, 2006.

[FW07] Gordon Fraser and Franz Wotawa. Nondeterministic testing with linear model-checker counterexamples. In *Quality Software, 2007. QSIC '07. Seventh International Conference on*, pages 107–116, 11-12 Oct. 2007.

[FWW08a] Gordon Fraser, Martin Weiglhofer, and Franz Wotawa. Coverage based testing with test purposes. In *Proceedings of the 8th International Conference on Quality Software*, pages 199–208, 2008.

[FWW08b] Gordon Fraser, Martin Weiglhofer, and Franz Wotawa. Using observer automata to select test cases for test purposes. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, pages 709–714, 2008.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *Proceedings of th 6th International Conference on Theory and Practice of Software Development*, volume 915 of *LNCS*, pages 82–96. Springer, 1995.

[GHN93] Jens Grabowski, Dieter Hogrefe, and Robert Nahm. Test case generation with test purpose specification by MSC's. In *Proceedings of the 6th SDL Forum*, pages 253–266. Elsevier Science, 1993.

[GLM02] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24, 2002.

[Ham77] R.G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

[HBH08] Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors. *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*. Springer, 2008.

[HFT00] Lex Heerink, Jan Feenstra, and Jan Tretmans. Formal test automation: The conference protocol with phact. In Hasan Ural, Robert L. Probert, and Gregor von Bochmann, editors, *Proceedings of the 13^{th} International Conference on Testing Communicating Systems*, volume 176 of *IFIP Conference Proceedings*, pages 211–220. Kluwer, 2000.

[HHT96] D. Hogrefe, S. Heymer, and G. J. Tretmans. Report on the standardization project "formal methods in conformance testing". In B. Baumgarten, H-J. Burkhardt, and A. Giessler, editors, *Selected proceedings of the IFIP TC6 9th international workshop on Testing of communicating systems*, pages 289–298, London, 1996. Chapman & Hall.

[HLU03] Olaf Henniger, Miao Lu, and Hasan Ural. Automatic generation of test purposes for testing distributed systems. In Alexandre Petrenko and Andreas Ulrich, editors, *3rd International Workshop on Formal Approaches to Testing of Software*, volume 2931, pages 178–191, Montreal, Quebec, Canada, 2003. Springer.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HP08] Jiale Huo and Alexandre Petrenko. Transition covering tests for systems with queues. *SOFTWARE TESTING, VERIFICATION AND RELIABILITY*, 18(4), 2008.

[HSSR99] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session initiation protocol. RFC 2543, IETF, 1999.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[IJW97] ITU-T SG 10/Q.8 ISO/IEC JTC1/SC21 WG7. Information retreival, transfer, and management for osi. framework: Formal methods in conformance testing. Committee Draft CD 13245-1, ITU-T Proposed Recommendation Z.500, 1997. Geneve, Switzerland.

[ISO89] ISO. ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989.

[JJ05] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, August 2005.

[KV96] Orna Kupferman and Moshe Y. Vardi. Module checking. In Rajeev Alur and Thomas A. Henzinger, editors, *8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 1996.

[LvBP94] Gang Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-statemachines using a generalized wp-method. *Transactions on Software Engineering*, 20(2):149–162, 1994.

[LY96] David Lee and Mihakus Yannakakis. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer, 1980.

[Mil90] Robin Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, chapter 19, pages 1201–1242. Elsevier Science Publishers B.V., 1990.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., 1979.

[OBY02] Vadim Okun, Paul E. Black, and Yaacov Yesha. Testing with model checker: Insuring fault visibility. In *Proceedings of International Conference on System*

Science, Applied Mathematics & Computer Science, and Power Engineering System, pages 1351–1356, 2002.

[PBG04] Alexandre Petrenko, Sergiy Boroday, and Roland Groz. Confirming configurations in efsm testing. *IEEE Transactions on Software Engineering*, 30(1):29–42, 2004.

[Phi87] I. Phillips. Refusal testing. *Theor. Comput. Sci.*, 50(3):241–284, 1987.

[PP05] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. *Electr. Notes Theor. Comput. Sci.*, 116:59–71, 2005.

[PPW$^+$05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering*, pages 392 – 401, St. Louis, Missouri, USA, 2005. ACM.

[PY02] Alexandre Petrenko and Nina Yevtushenko. Queued testing of transition systems with inputs and outputs. In Rob Hierons and Thierry Jéron, editors, *Proceedings of the Workshop Formal Approaches to Testing of Software*, pages 79–93, 2002.

[PY05] Alexandre Petrenko and Nina Yevtushenko. Conformance tests as checking experiments for partial nondeterministic fsm. In *Proceedings of the 5th International Workshop on Formal Approaches to Software Testing*, volume 3997 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2005.

[RdBJ00] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Proceedings of the Second International Conference on Integrated Formal Methods*, volume 1945 of *LNCS*, pages 338–357. Springer, 2000.

[RSC$^+$02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session initiation protocol. RFC 3261, IETF, 2002.

[SCSP03] Thitima Srivatanakul, John A. Clark, Susan Stepney, and Fiona Polack. Challenging formal specifications by mutation: a csp security example. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference*, pages 340–350. IEEE, December 2003.

[SNLD02] Henning Schulzrinne, Sankaran Narayanan, Jonathan Lennox, and Michael Doyle. SIPstone - benchmarking SIP server performance. Technical report, Columbia University, Ubiquity, 2002.

[SRS01a] Goran Stojsic, Robert Radovic, and Sinisa Srbljic. Formal definition of SIP end systems behavior. *EUROCON, Trends in Communications*, 2:293–296, 2001.

[SRS01b] Goran Stojsic, Robert Radovic, and Sinisa Srbljic. Formal definition of SIP proxy behavior. *EUROCON Trends in Communications*, 2:289–292, 2001.

[Sto93] Philip Alan Stocks. *Applying formal methods to software testing.* PhD thesis, Department of computer science, University of Queensland, 1993.

[TB03] Jan Tretmans and Ed Brinksma. TorX: Automated model based testing. In A. Hartman and K. Dussa-Zieger, editors, *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 13–25, Nurnburg, Germany, 2003.

[TPHT96] Rinke Terpstra, Luis Ferreira Pires, Lex Heerink, and Jan Tretmans. Testing theory in practice: A simple experiment. Technical report, University of Twente, The Netherlands, 1996.

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, December 1992.

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

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In Hierons et al. [HBH08], pages 1–38.

[VCG$^+$08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.

[Wei06] Martin Weiglhofer. A LOTOS formalization of SIP. Technical Report SNA-TR-2006-1P1, Competence Network Softnet Austria, Graz, Austria, December 2006.

[WLS04] Christian Wieser, Marko Laakso, and Henning Schulzrinne. SIP robustness testing for large-scale use. In *SOQUA/TECOS*, volume 58 of *LNI*, pages 165–178, 2004.

[WW08a] Martin Weiglhofer and Franz Wotawa. "On the fly" input output conformance verification. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 286–291, Innsbruck, Austria, February 2008.

[WW08b] Martin Weiglhofer and Franz Wotawa. Random vs. scenario-based vs. fault-based testing: An industrial evaluation of formal black-box testing methods. In *Proceedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 115–122, Funchal, Madeira - Portugal, 2008.