Protocol Conformance Testing a SIP Registrar: an Industrial Application of Formal Methods

Bernhard K. Aichernig\textsuperscript{1}  Bernhard Peischl\textsuperscript{2}  Martin Weiglhofer\textsuperscript{2}  Franz Wotawa\textsuperscript{1} *

\textsuperscript{1}Institute for Software Technology
Technische Universität Graz
8010 Graz, Austria
\{aichernig, wotawa\}@ist.tugraz.at

\textsuperscript{2}Competence Network Softnet Austria
Institute for Software Technology
Technische Universität Graz
8010 Graz, Austria
\{peischl, weiglhofer\}@ist.tugraz.at

Abstract

Various research prototypes and a well-founded theory of model based testing (MBT) suggests the application of MBT to real-world problems. In this article we report on applying the well-known TGV tool for protocol conformance testing of a Session Initiation Protocol (SIP) server. Particularly, we discuss the performed abstractions along with corresponding rationales, and exemplify our novel idea in designing test purposes by relying on structural and fault-based techniques. We present first empirical results obtained from applying our test cases to a commercial implementation and a popular open source implementation of a SIP Registrar. Notably, in both implementations our IOLTS Labeled Transition System model proved successful in revealing severe violations of the protocol.

1. Introduction

Today’s software and software-enabled systems are becoming increasingly complex, distributed and highly reactive. Due to this, the quality requirements in terms of the software product’s functional correctness are a major concern. Achieving functional correctness w.r.t. a given specification includes establishing appropriate software engineering methods, verification-, and validation techniques. Software testing, if carried out systematically and well-founded, is nowadays considered as an important task during the software life-cycle. However, in a practical setting, designing appropriate test cases is regarded as a difficult, tedious and thus rather expensive task.

The area of specification-based testing has made considerable advances in the recent years and research prototypes relying on various underlying representations are available today. Case studies report on successful application of concrete techniques to industrial sized applications [7, 16].

This article focuses on protocol conformance testing of the so called session initiation protocol (SIP) Registrar in the context of a commercial voice-over IP (VoIP) server \textsuperscript{1}. Due to the well-founded theory behind testing of input output labeled transition systems (IOLTS), the ability of this theory to be used with incomplete specification and due to the availability of mature research prototypes, we decided to use IOLTSs. For the generation of protocol conformance tests we rely on the TGV [15] tool, which is part of the CADP-toolbox [9]. TGV uses LOTOS as its primary input language, however, any other input language that provides IOLTS semantics can be applied alike.

This article contributes to the field of testing reactive systems in several aspects: We discuss critical issues regarding (problem-tailored) abstractions and, unlike to the majority of conducted case studies, provide rationales for these abstractions. Moreover, we outline how to obtain reasonable test purposes. Rather than relying on a single strategy, we propose two orthogonal strategies supplementing each other. First, we design test purposes by targeting structural coverage on our protocol formalization. Second, we applied the fault-based approach presented in [1], and report on scalability issues in applying this strategy to our industrial application. To overcome these intricacies we propose a novel extension to fault-based test purpose design. Moreover, we present an empirical evaluation and report on the typical errors found.

This paper continues as follows: in Section 2 we briefly introduce the input output conformance relation. In Section 3 we describe the SIP Registrar, which we use for the empirical evaluation. A classification system of abstractions, * Authors are listed in alphabetical order.

\textsuperscript{1}We conducted the research reported in this article in collaboration with the Kapsch CarrierCom AG, a leading provider of telecommunications infrastructure in Austria. Note that any information on the concrete product is to be treated confidentially and might require anonymization for final publication.
a coarse overview of our formal model and details about our abstractions are presented in Section 4. In Section 5 we discuss structured test purpose design and show how to apply fault-based test purpose generation to specifications with huge state spaces. In Section 6 we present empirical results and in Section 7 we discuss related work. Finally, in Section 8, we present our conclusions.

2. Preliminaries

TGV generates test cases in order to test input output conformance of an implementation with respect to an IOLTS model. Thus, we briefly review the input output conformance relation.

2.1. Input Output Conformance

In this section we introduce the models for test case generation and explain how they are used to describe specifications, implementations, test cases and test purposes. For a detailed discussion of the testing theory we refer to [23].

Definition 1 (Input Output LTS (IOLTS)) An IOLTS is a labeled transition system (LTS) $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ with $Q^M$ a finite set of states, $A^M$ a finite alphabet (the labels) partitioned into three disjoint sets $A^M = A^M_\text{l} \cup A^M_\text{o} \cup \{\tau\}$ where $A^M_\text{l}$ and $A^M_\text{o}$ are input and output alphabets and $\tau \not\in A^M_\text{l} \cup A^M_\text{o}$ is an unobservable, internal action, $\rightarrow_M \subseteq Q^M \times A^M \times Q^M$ is the transition relation and $q_0^M \in Q^M$ is the initial state.

We use the following classical notations of LTSs for IOLTSs. Let $q, q', q_\infty, b, q_\infty' \in Q^M$, $\sigma \in Q^M$ and $\sigma_0 \in A^M_\text{l} \cup A^M_\text{o}$.

- $q \xrightarrow{a} q'$ if $(q, a, b) \in \rightarrow_M$.
- $q \xrightarrow{b} q'$ if $(q, b, q') \in \rightarrow_M$.
- $q \xrightarrow{\tau} q'$ if $q \xrightarrow{\tau}$.
- $q \xrightarrow{\sigma} q'$ if $(q, \sigma, q') \in \rightarrow_M$. We denote $q \xrightarrow{\sigma}$ if $q \xrightarrow{\sigma_0}$.

$\rightarrow$ is used to represent quiescence.

We define $Out_M(q) = \{ a \in A^M_\text{O} | q \xrightarrow{a} \}$. $Out_M(q)$ is complete.

Commonly the symbol $\delta$ is used to represent quiescense. A quiescent state is a state that has no edge labeled with an output or an internal action. Thus, $q \xrightarrow{\delta}$ means that $q$ is a quiescent state. A LTS $M$ is called strongly responsive if it always eventually enters a quiescent state, that is $\forall q \in Q^M, \exists \omega : q' \in q \xrightarrow{\delta}.\omega$.

Note, that strongly responsive labelled transition systems do not have infinite loops labelled with the internal action $\tau$. We say, a LTS $M$ is strongly input enabled if it accepts every input in every state: $\forall a \in A^M_\text{l}, \forall q \in Q^M : q \xrightarrow{a}$.

$M$ is weakly input enabled if it excepts either an internal action or all input actions in all states: $\forall a \in A^M_\text{l}, \forall q \in Q^M : q \xrightarrow{a}$.

In order to define the input output conformance relation we need the set of suspension traces which is defined as $Straces(q) = \{ \sigma \in (A^M_\text{l} \cup A^M_\text{o})^* | q \xrightarrow{\sigma} \}$. For the input output conformance relation, we assume that the behavior of an implementation can be expressed by an IOLTS. This is called the test hypothesis.

Definition 2 (Input Output Conformance Relation (ioco)) The ioco relation says, that an implementation under test (IUT) conforms to a specification ($S$), if and only if the outputs of the IUT are outputs of $S$ after a suspension trace of $S$. Let $IUT = (Q^{IUT}, A^{IUT}, \rightarrow_{IUT}, q_0^{IUT})$ be weakly input enabled with $A^{IUT} = A^M_\text{l} \cup A^M_\text{o} \cup \{\tau\}$ and $S = (Q^S, A^S, \rightarrow_S, q_0^S)$ be strongly responsive with $A^S = A^M_\text{l} \cup A^M_\text{o}$.

$\in IUT \sigma S = \forall \sigma \in Straces(S)$

\[Out_{IUT}(\sigma) \subseteq Out_S(\sigma) \] (1)

Given $V \subseteq A^M_\text{l}$, hide $V$ in $M$ transforms a LTS $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ to another LTS $M' = (Q'^M, A'^M, \rightarrow_M, q_0'^M)$ where $Q'^M = Q^M$, $A'^M = A^M \setminus V$, $q_0'^M = q_0^M$ and $\forall (q, a, q') \in \rightarrow_M : (a \in V \Rightarrow (q, a, q') \notin \rightarrow_M \land (q, q') \in \rightarrow_{M'} \land \forall (q, b, q') \in \rightarrow_M : (b \in V \Rightarrow (q, b, q') \in \rightarrow_{M'})$.

2.2. Test Purposes

While formal specifications are descriptions of the system under test, a test purpose describes the test objectives for a test. Test purposes can be seen as a formal specification of a test case. In conformance testing the notation of a test purpose has been standarized [13]:

Definition 3 (Test purpose, informal) A description of a precise goal of the test case, in terms of exercising a particular execution path or verifying the compliance with a specific requirement.

Tools like SAMSTAG [10], TGV [15] and Microsoft’s xRTT [11] use test purposes for test generation. The formal notation of test purposes for TGV is given by:

Definition 4 (Test purpose, formal) Given a specification $S$ in form of an IOLTS, a test purpose is a deterministic IOLTS $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ equipped with two sets of sink states, Accept$^{TP}$ which defines Pass verdicts and Refuse$^{TP}$ which allows to limit the exploration of the graph $S$. Furthermore, $A^{TP} = A^S$ and $TP$ is complete ($\forall q \in Q^{TP}, a \in A^{TP} : q \xrightarrow{a}$).
According to [15] test synthesis within TGV is conducted as follows. Given a test purpose TP and a specification S, TGV calculates the synchronous product SP = S × TP. Afterwards, the visible behavior of SP is extracted by adding suspension labels and applying determination to SP, which leads to SPVIS. The determination removes internal actions τ from the synchronous product. SPVIS is equipped with AcceptVIS and RefuseVIS sink states. TGV derives a complete test graph from SPVIS by inverting inputs and outputs. States where an input is possible are completed for all other inputs and the verdicts pass, inconclusive and fail are assigned to the states.

2.3. Test Graphs and Test Cases

A test graph generated by TGV, using the algorithm outlined in Section 2.2, contains all test cases corresponding to a test purpose. Except for controllability the test graph already satisfies the properties of a test case TC = (QTC, ATC, →TC, q0TC) which are (cf. [1, 15]):

1. A test case only contains states from SPVIS and verdict states: QTC ⊆ QVIS ∪ (fail, pass, inconc), and q0TC = q0VIS.
2. TC mirrors image of actions and considers all possible outputs of the IUT: ATC = AITC ∪ AO with AITC ⊆ AIT and AO ⊆ AOVIS.
3. From each state a verdict must be reachable: ∀q, ∃σ ∈ TC → TC: (q ∈ {fail ∪ inconc ∪ pass}) ⇒ q →TC q′.
4. States in fail and inconc are only directly reachable by inputs: ∀(q, a, q′) ∈ →TC: (q′ ∈ {fail ∪ inconc}) ⇒ a ∈ AITC).
5. A test case is input complete in all states where an input is possible: ∀q ∈ QTC : (∃a ∈ AITC, q →TC q ⇒ ∀b ∈ AOTC, q →TC q).
6. TC is controllable: no choice is allowed between two outputs or between inputs and outputs: ∀q ∈ QTC, ∀a ∈ AOTC : q →TC ⇒ ∀b ∈ AITC ∪ AOTC \ {a} : q →TC.

3. SIP Registrar

The Session Initiation Protocol (SIP) 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. Essentially, 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 so called Registrar, which is responsible for maintaining location information of users.

SIP uses a request/response transaction model. Messages are encoded in UTF-8, i.e., SIP is a text based protocol. A message consists of a start-line, a message-header and a message-body. The start-line indicates the request method or the type of response. In its basic version SIP defines six different request methods. One of them is the REGISTER method, which associates a user address with an end point. This is the main method for the Registrar.

The message-header of a SIP message contains information like the originator, the recipient, and the content-type of the message. A REGISTER messages may contain CONTACT header fields which are used to modify stored user location information. In the case of the SIP Registrar the message bodies are usually empty.

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 is encrypted with the HTTP-Digest method [8]. This request is accepted by the Registrar and answered with “200 OK”. For a full description of SIP we refer to [14].

4. Formal Specification and Level of Abstraction

The level of abstraction is determined by the objectives of the model. The aim of our formal specification is protocol conformance testing. This specification objective requires that abstractions do not omit details which are essential for testing. Especially in the context of an industrial
application, the level of abstraction is a crucial property of a formal model. If abstractions discard important details, the error detection capability of the formal model decreases significantly. On the other hand, if the model reflects all details from the concrete world there might be a huge number of possible (redundant) test cases or even the generation of test cases becomes infeasible.

### 4.1. Classification of Abstractions

According to [18, 19] we distinguish 5 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 abstractions deal 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 abstractions combine different real world aspects into logical units within the model.

In terms of a sixth category we propose to extend the classification of abstractions by adding environmental assumptions. This specific category subsumes assumptions about the test environment simplifying the formal model. For example, the assumption that test messages are delivered reliably and in the send order falls into that category.

### 4.2. Abstractions for the SIP-Registrar

We derived the formal specification from a textual document, namely the RFC 3261 [14], which specifies the Session Initiation Protocol. Textual descriptions typically suffer from ambiguity. Especially the particular keywords of an RFC, e.g., MAY and SHOULD [3], introduce some implementation freedom. To be able to check any implementation for conformance with the specification, the model must reflect optional parts of the specification. That is, tests derived from the specification should not reject implementations that do not implement optional parts. Implementations should only be marked as erroneous if an optional feature is implemented wrong.

As illustrated in Figure 2, our Registrar model comprises two main processes. One process models the server transaction that is used to handle incoming and outgoing messages (serverTransactionInterface). The other process contains the logic for handling register requests (registrarCore).

The registrarCore process uses two global variables which hold information about authorized users and about the contact information of the users. The process processRegister is invoked for every incoming REGISTER messages. It uses the set of configured users, cfg, to determine if the user is allowed to modify the contact information. The set of registered contacts, reg, is updated according to the contact information given by the REGISTER message. The formal model communicates with its environment through the two gates (ports) pin and pout. A detailed discussion of the specification, including the full LOTOS source, can be found in [24].

Table 1 lists the abstractions of our SIP Registrar model. We abstract from general server errors (Abstraction 1) because of the loose informal specification of server errors within the RFC. Server errors may occur any time, when the Registrar encounters an internal error. For testing general server errors we would need a high knowledge about the implementation internals. Especially we need to know how to enforce server errors during test execution.

Abstraction 2 omits specification details about forwarding REGISTER requests. Thus we do not generate tests for this feature. We also skipped the REQUIRES header field in the formal specification in order to limit the number of possible request messages (Abstraction 4).

Abstracting from the calculation of authentication credentials (Abstraction 3) does not impose any limitation if the credentials are calculated and inserted correctly into test messages during test execution.

Abstractions 5-8 are based on the ideas of equivalence partitioning and boundary value analysis [17], which are strategies from classical white-box testing. For example, Abstraction 8 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. Thus, we only need three different users, one of each group.

Abstraction 9 limits the different CONTACT header field
Table 1. Abstractions for the specification of the SIP Registrar.

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>func-</td>
<td>Our formal model of the Registrar does never terminate with a server error.</td>
</tr>
<tr>
<td></td>
<td>tional</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>func-</td>
<td>Our specification never forwards REGISTER messages to other SIP Registrars.</td>
</tr>
<tr>
<td></td>
<td>tional</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>func-</td>
<td>While the authentication handshake is in our specification, the calculation of authentication credentials is not modelled.</td>
</tr>
<tr>
<td></td>
<td>tional</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>data</td>
<td>REGISTER messages do not contain any REQUIRED header fields.</td>
</tr>
<tr>
<td>5</td>
<td>data</td>
<td>The CALL-ID is abstracted to the range [0, 1].</td>
</tr>
<tr>
<td>6</td>
<td>data</td>
<td>We limit the integer part of the CSEQ header to [0, 1]. The method part is not in the formal model.</td>
</tr>
<tr>
<td>7</td>
<td>data</td>
<td>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.</td>
</tr>
<tr>
<td>8</td>
<td>data</td>
<td>Our model distinguishes three different users.</td>
</tr>
<tr>
<td>9</td>
<td>data</td>
<td>The formal specification of the Registrar distinguishes three different CONTACT values: *, any_addr1, and any_addr2.</td>
</tr>
<tr>
<td>10</td>
<td>data</td>
<td>The TO and FROM header fields are omitted in our abstract REGISTER messages.</td>
</tr>
<tr>
<td>11</td>
<td>temp-</td>
<td>Our specification does not use any timers. We only focus on the ordering of events.</td>
</tr>
<tr>
<td>12</td>
<td>env.</td>
<td>We assume that the communication channel is reliable and delivers messages in the sent order.</td>
</tr>
<tr>
<td>13</td>
<td>env.</td>
<td>For every test case, the Registrar starts from a well known initial state.</td>
</tr>
</tbody>
</table>

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.

Abstraction 2 causes the header fields, TO and FROM, to contain redundant information. So they can be omitted from our formal REGISTER messages (Abstraction 10).

As TGV does not support real-time testing, we need to abstract from concrete timer events (Abstraction 11).

Assumption 12 is ensured during test execution by running the test execution framework and the implementation under test on the same computer. A reset of the system under test before running a single test guarantees that Assumption 13 holds for every test.

5. Finding Test Purposes

TGV uses test purposes to specify the test objective for a certain test suite. By the use of these test purposes, TGV allows the generation of test cases without constructing the labeled transition system for the whole specification. Test purpose design may become rather complex. For example, in [6] the authors tried to detect mutated versions of an implementation. Even after ten hours of test purpose design, they did not manage to come up with a set of test purposes that detects all faulty mutants.

As our experience indicates, in practice, under presence of a mature test process, we typically encounter two orthogonal and thus supplementing test-design practices. First, test cases are designed to cover some kind of structural coverage criteria on the source code, like, for example, MC/DC coverage [17]. Second, test engineers or developers often anticipate defects relying on their domain knowledge, their intuition, or on errors previously made [2]. This strategy is very effective, if sound domain knowledge is available and major standards for developing safety-critical software (namely the MISRA standard, ISO-61508 and EN-50128) recommend such techniques supplementary to techniques relying on structural coverage.

Conceptually, we thus propose to employ a similar strategy even for the design of reasonable test purposes: Our novel idea relies on both, to derive the test purpose from the specification’s structural properties and from the anticipated fault models. This orthogonal strategy, which has proved successful for test case design, may also bring considerable benefits for test purpose design.

5.1. Structural Test Purpose Design

In the case of the SIP-Registrar we used condition/decision coverage on our formal model. In order to avoid many test cases that detect the same errors, test purposes should be orthogonal. That is, the number of edges from the specification that are selected by several test purposes should be minimal.

Typically TGV is rather efficient on test purposes, that select only a small set of edges in states that are not close to the accept and the refuse states, than on test purposes that allow many edges in that states. For large specifications, it appears to be a reasonable strategy to have many edges that lead to refuse states.

5.2. Fault-based Test Purpose Design

Applying the idea of fault-based test case design to the development of test purposes leads to the approach presented in [1].

Basically, the idea is to prevent the implementation under test to conform to a faulty specification. Therefore the authors of [1] use mutation operators in order to generate faulty mutants from the original specification. They generate the IOLTS $S_e$ for the original specification and the IOLTS $S^M_e$ for every mutant $M$. Afterwards $S_e$ and $S^M_e$ are minimized to $S$ and $S^M$, respectively. An equivalence check gives a discriminating sequence $c$ if there is an observable difference between $S$ and $S^M$. This sequence $c$,
possibly extended by one more valid transition, is used as test purpose \( tp \). The use of test purpose \( tp \) on \( S \) gives a test case that fails if the implementation conforms to the faulty specification \( S^M \).

This approach has been applied successfully for testing the Apache HTTP server where it revealed an unexpected behavior of the server. However, currently this approach cannot be applied to models with huge state spaces. The reason for this limitation is that the CADP toolbox is currently unable to translate large (LOTOS) models to IOLTSs. In case of our SIP Registrar model CADP runs out of memory (2 GB) after 11 days. Hence, the equivalence check between the original and the mutated IOLTS model cannot be performed.

### 5.2.1 Coping with Large Specifications

Since the CADP toolbox is unable to translate large LOTOS models into IOLTS, it is necessary to extract a slice from the specification that includes the relevant parts only. The relevant part for our equivalence check are the places where we know where the specification has been mutated. Hence, the fault has been introduced in the mutant. Fortunately, the test processes that reflect the relevant behavior of their models. Hence, the size of the model does not matter any more since the equivalence check is performed on the test processes. An example serves to illustrate this technique.

Figure 3 illustrates the application of the event swap mutation operator to a LOTOS specification. The order of the two events \( g_2 \) and \( g_1 \) in Line 3 has been changed from \( g_2; g_1 \) (original) to \( g_1; g_2 \) (mutant). Both versions of the specification have been annotated with \( \alpha \) and \( \beta \). Note, that \( \alpha \) and \( \beta \) are not in the language of the original specification \( L \), i.e. \( \{ \alpha, \beta \} \not\subseteq A^L \). The labeled transition systems described by the specification and the mutant are depicted in Figure 4.

By the use of a test purpose, that accepts traces that end in \( \alpha \), but refuses traces that contain \( \beta \) we extract a test graph that includes the fault induced by the specific mutation. Figure 5 illustrates the used test purpose and the extracted test graph. Note, that this figure only shows the test graph of the mutant. The test graph for the original specification looks similar, except that it has the correct ordering of \( g_1 \) and \( g_2 \).

Now we hide \( \alpha \) and \( \beta \) in the test graphs, i.e., we transform \( \alpha \) and \( \beta \) to the internal event \( \tau \). Calculating the discriminating sequence (using CADP-Bisimulator) between the two test graphs leads to \( g_1; g_2 \). This is our new test purpose which is used on the original specification.

Formally, we generate a test purpose for a specification \( L = (Q^L, A^L, \rightarrow_L, q_0^L) \) as follows:

1. Select a mutation operator \( O_m \).
2. Use the knowledge where \( O_m \) changes the specification, to generate \( L' \) by inserting markers \( \{\alpha, \beta\} \not\subseteq A^L \) into the formal (LOTOS) specification \( L \).
3. Generate a mutated version of the specification \( L^m = O_m(L') \) by, applying \( O_m \) to the marked, formal specification \( L' \).
4. Generate two complete test graphs, $CTG_α$ for the specification and $CTG_α^{m}$ for the mutant, by the use of the test purpose from Figure 5 (using CADP-TGV).

5. Hide the additional added labels by transforming them to internal transitions $τ$ (using CADP-Beg). This leads to $CTG_α′ = hide α, β$ in $CTG_τ$ and to $CTG_α^{m}′ = hide α, β$ in $CTG_τ^{m}$.

6. Minimize $CTG_α′$ and $CTG_α^{m}′$ using the Safety Equivalence relation in order to obtain $CTG$ and $CTG^{m}$ (using CADP-Reducer).

7. Check $CTG$ and $CTG^{m}$ for Strong Bisimulation (using CADP-Bisimulator). The counterexample $c$, if any, gives the new test purpose. $c$ is extended by a valid transition (if any) in order to create a valid path which discovers the injected error.

8. Generate a test case from the new test purpose (using CADP-TGV).

Note, that a mutation operator might change the specification in a way, that $α$ cannot be reached from $q_0^{L^m}$. In that case any sequence $c$ of $CTG_τ$ that ends in $α$ is a discriminating sequence. By hiding $α$ and $β$ in $c$ and possibly adding one more valid transition, we obtain our new test purpose.

This approach capabilities depend on the insertion strategy of the markers $α$ and $β$. For our evaluation we use a strategy that directly inserts $α$ subsequently to the position of the scheduled mutation. For example, if we use the event swap operator and swap the events $e_i, e_k$ then we add $α$ after the second event. While the mutant contains $e_k, e_i, α$ the original specification looks like $e_i, e_k, α$. Note, that this currently restricts our approach to mutations where the introduced fault is observable at the position of the mutation. That is, a mutation must not effect internal ($τ$) transitions.

We insert $β$ as first event in every process except $p_α$, which is the process that contains $α$. Additionally, we insert $β$ in all branches of $p_α$ that do not contain $α$. This insertion rule arises the problem, that $α$ may not be reachable anymore. For example, assume that we insert $α$ and $β$ in a recursive process. The generation of a test graph fails, if the execution of the block that contains $α$ depends on previous executions of blocks that contain $β$.

To overcome that problem we use the test purpose to stepwise unroll dependencies of marked blocks. If $TGV$ generates an empty test graph for the test purpose of Figure 5 and the un-mutated marked specification we extend the test purpose. The new test purpose, illustrated in Figure 6, allows one $β$ transition before $α$. If $TGV$ fails to produce a test graph with the new test purpose, we again add one $β$ transition to the test purpose. This procedure is continued until a test graph can be generated.

As depicted in the next section, first experiments using this novel technique show promising results.

6. Empirical Evaluation

In this section we present our results obtained from executing the generated test cases against a commercial SIP Registrar and against OpenSER, an open source SIP Registrar.

For test purposes designed according to the structural approach of Section 5.1, we extract the test cases from the complete test process (graph), that is generated by $TGV$. If a test graph contains loops, it describes infinitely many test cases. Thus, we use a heuristic for the extraction of our test cases. A detailed discussion of our extraction algorithm can be found in [25].

In the case of fault-based test purposes we use a single test case generated by $TGV$ for every test purpose, because every test case derived from such a test purpose reveals the fault.

Independently of the used test purpose design approach, the produced test cases are abstract test cases. The transitions of an abstract test case describe stimuli and expected responses of the implementation under test in an abstract manner. During test execution stimuli are refined to concrete protocol messages, while system responses are transformed to an abstract representation. Details of our test execution framework can be found in [25].

6.1. SIP Registrar Specification

In cooperation with the industry partner’s domain experts we developed a formal specification covering the full functionality of a SIP Registrar. This LOTOS specification consists of appr. 3KLOC (net.), 20 data types (contributing to net. 2.5KLOC.), and 10 processes. Note, that the Registrar determines response messages through evaluation of the request data fields rather than using different request messages. Thus, our specification heavily uses the concept of abstract data types.

A structured review of this specification considerably increased our confidence in the formalization. Particularly,
the performed abstractions can be considered as thoroughly reviewed w.r.t. the model’s fault detection capabilities as the industry partner’s domain experts required rationale for critical issues. To our best knowledge, the developed specification thus represents a correct formalization of the SIP Registrar.

6.2. Results for Structural Test Purposes

Table 2 lists the number of test cases being generated (2nd column), the running time of the TGV tool (3rd column), the time required for minimizing the IOLTS (with CADP-Bcg) using branching equivalence (4th column), and the amount of time it takes to apply our test case extraction (5th column) for those test cases associated with a certain test purpose (1st column).

By relying on our model, TGV creates the majority of test cases in a reasonable amount of time, however, for the delete test purpose, we obtain a significant outlier. This specific test purpose captures scenarios in which a client registers at the server, but removes its registration subsequently. Due to the various possibilities of registration scenarios the test purpose results in a rather complex structure causing the substantial increase in running time for creating the corresponding test graph.

Table 3 outline the results of executing the obtained test cases against the commercial and the OpenSER Registrar in terms of the number of executed (2nd column), passed (3rd and 5th column), and failed test cases (4th and 6th column).

Our specification proofed detailed enough to reveal severe misbehavior and, as exemplified in terms of the OpenSER SIP Registrar, general enough to apply it to an arbitrary SIP Registrar implementation. The high number of failed tests on both products are due to overlapping test purposes (see Section 5.1) and overlapping test cases, which cover equivalent faults.

For the commercial SIP Registrar we discovered 9 different faults. In the case of the OpenSER SIP Registrar we found 4 discrepancies between the implementation and the specification.

Most of the detected errors (commercial: 6, OpenSER: 2) are regarding different combinations of CONTACT header fields. For example, the Registrar deletes all stored contacts for a certain SIP-Uri if a message contains a delete request (EXPIRES: 0; CONTACT: *) combined with a standard regular contact header (CONTACT: 10.0.0.1). According to the RFC such a request should be rejected with “Bad Request”.

The RFC specifies, that each stored contact has to record the CALL-ID value and the CSEQ number of the request that created the contact information. Stored contact information should be removed only if the CALL-ID value of the delete request differs from the stored CALL-ID or if the CSEQ value of the delete request is higher than the stored value. Both tested implementations violate this property, because they delete stored contact information also if the CALL-ID numbers and the CSEQ values are equal.

Another detected error that causes many test cases to fail, is the extension of the requested contact expiration interval. According to the RFC a Registrar may decrease the requested expiration interval, but it is not allowed to increase the requested interval. Both implementations possible increase a requested interval. This error causes test cases, generated by the interval too brief test purpose, to fail. Because our test purposes overlap this misbehavior is also detected by test cases of the register ok, of the delete, and of the unauthorized test suite.

We detected two errors, where the commercial SIP Registrar does not respond. The Registrar does not reject unknown users with “404 Not Found” if authentication is turned off. Because of this single error all test cases of the not found test suite fail. Furthermore, the commercial implementation does not reject messages with malformed CONTACT header fields correctly. Thus all tests of the invalid request test suite fail on the commercial implementation.

6.3. Results for Fault-Based Test Purposes

Due to the size of the Registrar specification and the huge number of possible mutants, we need to automate our fault-based test purpose generation to obtain results for all mutation operators of [1]. However, we chose three mutation operators (event insert operator (eio), event swap operator (eso), missing condition operator (mco)) and applied our approach manually.

Table 4 lists the details of our fault-based test purpose generation. This table shows the number of possible mu-
tants, i.e., mutants that can be generated by the mutation operator (2nd column) and the number of generated mutants (3rd column), i.e., mutants not influencing $\tau$ transitions (see Section 5.2.1). Note, that only for the event swap operator we encountered one mutant, where the fault is not visible directly at the position of the mutation. Additionally, we list for how many mutants we are able to generate test processes using our approach (4th column). The 5th column shows the number of equivalent mutants, i.e., there is no observable difference between the mutant and the specification.

Using our fault-based test purpose generation approach, we are able to generate 30, 7, and 32 test purposes out of 35, 9, and 46 mutants within reasonable time (6th column). 7 of the created mutants are equivalent to the original specification. For the other 21 mutants, TGV runs out of memory (2 GB) when generating the test graph. By having a closer look at that 21 mutants, we saw that the mutation affected parts of our specification where decisions are based on a certain internal state. The event sequence for establishing this internal state contains $14 \beta$. Since we run out of memory every time we allow $12 \beta$ before $\alpha$, we are currently unable to generate the test processes for that 21 mutants.

Although, the time until TGV runs out of memory (7th column) is very small compared to the time until CDPCaesar runs out of memory when we try to construct the IOLTS of the specification (11 days). Even if the generation of a test graph fails, because of lack of memory, the average time is acceptable. Finally, Table 4 lists the time needed by TGV to generate a test case for a generated test purpose.

Table 5 illustrates the results when executing the test cases derived from fault-based test purposes against OpenSER and against the commercial SIP Registrar. The table shows the number of test cases (2nd column), the number of passed, failed and inconclusive test cases for OpenSER (3rd, 4th and 5th column) and for the commercial Registrar (6th, 7th and 8th column). The tests were executed against the two implementations by having the authorization feature turned off (2nd-4th row) and by having the authorization turned on (5th-7th row).

These additional test cases revealed an additional error in the commercial SIP Registrar, which has not been detected by our test cases from structural test purposes. The implementation considers a message as a retransmission of a previous message although the BRANCH parameter fields differ. This behavior violates the RFC.

### 7. Related Research

Various testing techniques have been applied to SIP [26, 20]. While the applied techniques deal with security aspects and performance issues, to our best knowledge none of them focuses on protocol conformance testing.

Modelling SIP using SDL or UML has been subject to publication previously [4, 21, 22]. In difference to our formalization, the presented models are based on the outdated RFC 2543 [12]. The presented formalizations are not tailored to any special purpose and deal with the session management part of SIP. In difference to that, 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.

There exist various case studies on using TGV for automatic test generation. For example, the authors of [6] used TGV for test generation and TOrX for test execution. Tests were generated by the use of manually-designed test purposes and by the use of randomly-generated test purposes. The test cases were evaluated on 24 loco-incorrect mutants. Kahlouche et al. present in [16] the application of TGV to the cache coherency protocol, while the authors of [7] presented the application of TGV to the DREX protocol. The latter work compares the tests produced by TGV to handwritten tests. However, none of these case studies discusses test purpose design in detail. Previous publications do not present abstractions within their formal model and provide rationales for these abstractions. Furthermore, these case studies do not consider mutation testing.

### 8. Conclusion

In this article we point out various abstractions performed for developing an appropriate formal model and, unlike to many other articles reporting from the successful application of specification-based testing, we provide rationales for performing the chosen abstractions. Our industrial sized problem from the telecommunications area pursues two different strategies for test purpose design. One strategy relies on a structural coverage criteria whereas a sec-

<table>
<thead>
<tr>
<th>Operator</th>
<th>No. Mutants</th>
<th>Avg. Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>poss.</td>
<td>appl.</td>
</tr>
<tr>
<td>eio</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>eso</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>mco</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>Total</td>
<td>91</td>
<td>90</td>
</tr>
</tbody>
</table>

Table 4. Test generation time.

<table>
<thead>
<tr>
<th>Operator</th>
<th>commercial</th>
<th>OpenSER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pass</td>
<td>fail</td>
</tr>
<tr>
<td>eio</td>
<td>30</td>
<td>26</td>
</tr>
<tr>
<td>eso</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>mco</td>
<td>29</td>
<td>26</td>
</tr>
<tr>
<td>eio auth.</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>eso auth.</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>mco auth.</td>
<td>29</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>124</td>
<td>89</td>
</tr>
</tbody>
</table>

Table 5. Test execution results.
ond and orthogonal one exploits the knowledge on faults in terms of fault models. Particularly for the fault-based test purpose design, we encountered scalability problems and thus we propose a novel technique for fault-based test purpose design and also discuss first experiments on our real world application. In addition, we discuss the error detected on both, an open-source and an commercial voice over IP server. Notably, our derived test cases detect severe failures in both implementations. Using two different test purpose design strategies proved to be successful, since the fault-based method reveals an additional error that we miss with our structural designed test purposes.

However, our approach needs further evaluation. We need to automate the generation of mutants for LOTOS specifications in order to evaluate our approach for all mutation operators. Furthermore, we need to evaluate other marking strategies for $\alpha$ and $\beta$.

Our current formalization of the SIP Registrar makes heavy use of abstract data types. Thus, we need to evaluate the efficiency of our test purpose design approaches when using symbolic test generation techniques. Especially the use of STG [5], which uses symbolic transition systems instead of labeled transition systems, appears to be a promising alternative to TGV, at least for our data dependent Registrar specification.

References


