Fifth IEEE International Conference on Software Engineering and Formal Methods

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

Bernhard K. Aichernig[1] Bernhard Peischl[2] Martin Weiglhofer[2] Franz Wotawa[1] †

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

[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. Furthermore, we show how to use structural and fault-based techniques for test purpose design. We present first empirical results obtained from applying our test cases to a commercial implementation and to a popular open source implementation of a SIP Registrar. Notably, in both implementations our input output 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 various research prototypes are available today (e.g., [21, 22]). Case studies report on successful application of concrete techniques to industrial sized applications [7, 13].

This article focuses on protocol conformance testing of a session initiation protocol (SIP) Registrar in the context of a commercial voice-over IP (VoIP) server. Due to the well-founded theory behind testing of input output labeled transition systems (IOLTS) [21], 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 [12] tool, which is part of the CADP toolbox [8]. 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 use two different test purpose design strategies. 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 em-

*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 Wirtschaftsfrderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

†Authors are listed in alphabetical order.

0-7695-2884-8/07 $25.00 © 2007 IEEE
DOI 10.1109/SEFM.2007.29

IEEE computer society

pirical evaluation. A classification system of abstractions, 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 (ioco) of an implementation with respect to an IOLTS model. Thus, we briefly review the ioco 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 [21].

Definition 1 *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 $A^M = A_I^M \cup A_O^M \cup \{\tau\}$ where A_I^M and A_O^M are input and output alphabets and $\tau \notin A_I^M \cup A_O^M$ 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 $q, q', q_i \in Q^M, Q \subseteq Q^M, a_{(i)} \in A_I^M \cup A_O^M$ and $\sigma \in (A_I^M \cup A_O^M)^*$. Then $q \xrightarrow{a}_M q' =_{df} (q, a, q') \in \rightarrow_M$ and $q \xrightarrow{a}_M =_{df} \exists q' : (q, a, q') \in \rightarrow_M$, and $q \not\xrightarrow{a}_M =_{df} \nexists q' : (q, a, q') \in \rightarrow_M$. $q \stackrel{\epsilon}{\Rightarrow} q' =_{df} ((q = q') \vee (q \xrightarrow{\tau}_M q_1 \wedge \cdots \wedge q_{n-1} \xrightarrow{\tau}_M q'))$ and $q \stackrel{a}{\Rightarrow} q' =_{df} \exists q_1, q_2 : q \stackrel{\epsilon}{\Rightarrow}_M q_1 \xrightarrow{a}_M q_2 \stackrel{\epsilon}{\Rightarrow}_M q'$ which generalizes to $q \stackrel{a_1 \ldots a_n}{\Rightarrow} q' =_{df} \exists q_0, \ldots, q_n : q = q_0 \stackrel{a_1}{\Rightarrow}_M q_1 \ldots q_{n-1} \stackrel{a_n}{\Rightarrow}_M q_n = q'$. We denote q **after**$_M$ $\sigma =_{df} \{q' | q \stackrel{\sigma}{\Rightarrow}_M q'\}$ and Q **after**$_M$ $\sigma =_{df} \bigcup_{q \in Q}(q$ **after**$_M$ $\sigma)$. We define $Out_M(q) =_{df} \{a \in A_O^M | q \xrightarrow{a}_M\}$ and $Out_M(Q) =_{df} \bigcup_{q \in Q}(Out_M(q))$. We will not always distinguish between an IOLTS and its initial state and write $M \Rightarrow_M$ instead of $q_0^M \Rightarrow_M$. We will omit the subscript $_M$ (and superscript ^{M}) when it is clear from the context.

Commonly the symbol δ is used to represent quiescence. A quiescent state is a state, that has no edge labeled with an output or an internal action. Thus, $q \xrightarrow{\delta} q$ means, that q is a quiescent state. An IOLTS M is called *strongly responsive* if it always eventually enters a quiescent state. Note, that strongly responsive labelled transition systems do not have infinite loops labelled with the internal action τ. We say, an IOLTS M is *weak input enabled* if it accepts either an internal action or all input actions in all states: $\forall a \in A_I^M, \forall q \in Q^M : q \stackrel{a}{\Longrightarrow}$. An IOLTS is *deterministic* if for any trace there is at most one successor state, i.e., $\forall q \in Q^M, \forall \sigma \in \left(A_I^M \cup A_O^M\right)^* : |q$ **after**$_M \sigma| \leq 1$, where $|X|$ denotes the cardinality of the set X.

To define the input output conformance relation we need the suspension automaton, which makes quiescence observable by considering δ as an output.

Definition 2 *The suspension automaton of an IOLTS $M = \left(Q^M, A^M, \rightarrow_M, q_0^M\right)$ is an IOLTS $\Delta(M) = (Q^M, A^{\Delta(M)}, \rightarrow_{\Delta(M)}, q_0^M)$ where $A^{\Delta(M)} = A^M \cup \{\delta\}$ with $\delta \in A_O^{\Delta(M)}$. The transition relation $\rightarrow_{\Delta(M)}$ is obtained from $\rightarrow_M$ by adding loops $q \xrightarrow{\delta} q$ for each quiescent state. The traces of $\Delta(S)$ are called the suspension traces of S and are denoted by $Straces(S)$.*

For the ioco relation, we assume that the behavior of an implementation can be expressed by an IOLTS.

Definition 3 *The ioco relation says, 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. Let $IUT = \left(Q^{IUT}, A^{IUT}, \rightarrow_{IUT}, q_0^{IUT}\right)$ be weakly input enabled with $A^{IUT} = A_I^{IUT} \cup A_O^{IUT} \cup \{\tau\}$ and $S = \left(Q^S, A^S, \rightarrow_S, q_0^S\right)$ be strongly responsive with $A^S = A_I^S \cup A_O^S \cup \{\tau\}$. The alphabets of the IUT and S are compatible, i.e., $A_I^S \subseteq A_I^{IUT}$, and $A_O^S \subseteq A_O^{IUT}$. Then the ioco relation is defined as follows:*

$$IUT \textbf{ ioco } S =_{df} \forall \sigma \in Straces(S) :$$
$$Out_{IUT}(\Delta(IUT) \textbf{ after}_{IUT} \sigma) \subseteq Out_S(\Delta(S) \textbf{ after}_S \sigma).$$

If $V \subseteq A^M$ then $hide\ V\ in\ M =_{df} (Q, A^M \setminus V, \rightarrow, hide\ V\ in\ q_0^M)$ where $Q = \{hide\ V\ in\ q | q \in Q^M\}$ and $\rightarrow$ is the minimal set satisfying the following interference rules ($a \in A^M \cup \tau$):

$$q \xrightarrow{a} q', a \notin V \quad \vdash \quad hide\ V\ in\ q \xrightarrow{a} hide\ V\ in\ q'$$
$$q \xrightarrow{a} q', a \in V \quad \vdash \quad hide\ V\ in\ q \xrightarrow{\tau} hide\ V\ in\ q'$$

2.2. Test Purposes

While formal models are descriptions of the system under test, a test purpose describes the test objectives for a set of tests. Test purposes can be seen as a formal specification of a test case. Tools like SAMSTAG [9], TGV [12] and Microsoft's XRT [10] use test purposes for test generation. The formal notation of test purposes for TGV is given by:

Definition 4 (Test purpose) *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 trap 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* ($\forall q \in Q^{TP}, a \in A^{TP} : q \xrightarrow{a}_{TP}$).

According to [12] test synthesis within TGV is conducted as follows. Given a test purpose TP and a specification S TGV calculates the synchronous product $SP = S \times TP$. Afterwards, the visible behavior of SP is extracted by adding suspension labels and applying determinization to SP, which leads to SP^{VIS}. The determinization removes internal actions τ from the synchronous product. SP^{VIS} is equipped with $Accept^{VIS}$ and $Refuse^{VIS}$ sink states. TGV derives a *complete test graph* from SP^{VIS} by inverting outputs and inputs. States where an input is possible are completed for all other inputs and the verdicts *pass*, *inconc* (inconclusive) and *fail* are assigned to the states.

2.3. Test Graphs and Test Cases

In the *ioco* testing framework a test case is modeled as an IOLTS that synchronizes with the model of the IUT.

Definition 5 *A test case is a deterministic IOLTS* $TC = (Q^{TC}, A^{TC}, \rightarrow_{TC}, q_0^{TC})$ *equipped with three sets of trap states* $Pass \subset Q^{TC}$*,* $Fail \subset Q^{TC}$*, and* $Inconc \subset Q^{TC}$ *characterizing verdicts. A test case has to satisfy following properties:*

1. *A test case only contains states from* SP^{VIS} *and verdict states:* $Q^{TC} \subseteq Q^{VIS} \cup Pass \cup Inconc \cup Fail$*, and* $q_0^{TC} = q_0^{VIS}$.
2. TC *mirrors image of actions and considers all possible outputs of the IUT:* $A^{TC} = A_I^{TC} \cup A_O^{TC}$ *with* $A_I^{TC} \subseteq A_O^{IUT} \cup \{\delta\}$ *and* $A_O^{TC} \subseteq A_I^{VIS}$.
3. *From each state a verdict must be reachable:* $\forall q \in Q^{TC}, \exists \sigma \in A^{TC*}, \exists q' \in Pass \cup Inconc \cup Fail : q \stackrel{\sigma}{\Rightarrow}_{TC} q'$.
4. *States in* $fail$ *and* $inconc$ *are only directly reachable by inputs:* $\forall (q, a, q') \in \rightarrow_{TC}: (q' \in Inconc \cup Fail \Rightarrow a \in A_I^{TC})$.
5. *A test case is input complete in all states where an input is possible:* $\forall q \in Q^{TC} : (\exists a \in A_I^{TC}, q \xrightarrow{a}_{TC} \Rightarrow \forall b \in A_I^{TC}, q \xrightarrow{b}_{TC})$.
6. TC *is controllable, i.e., no choice between two outputs or between inputs and outputs:* $\forall q \in Q^{TC}, \forall a \in A_O^{TC} : q \xrightarrow{a}_{TC} \Rightarrow \forall b \in A^{TC} \setminus \{a\} : q \not\xrightarrow{b}_{TC}$.

A *test graph* generated by TGV contains all test cases corresponding to a test purpose. Except for controllability the test graph already satisfies the properties of a test case. A *test suite* is a set of test cases.

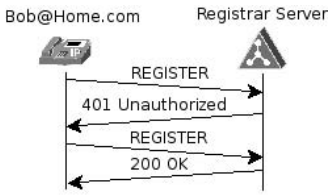

Figure 1. SIP registration example.

3. SIP Registrar

The Session Initiation Protocol (SIP) [17] handles communication sessions between two end points. The focus of SIP is the signaling part of a communication session independent of the 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. The Registrar is responsible for maintaining location information of users in a SIP network.

SIP is a text based protocol, that uses a request/response transaction model. 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. The main method for the Registrar, which is the REGISTER method, associates a user address with an end point. 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 of a 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 accepted by the Registrar and answered with "200 OK".

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 essen-

tial 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 [15, 16] 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* 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.

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 sent 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 [17], 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 be marked as erroneous only 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 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 the environment through the two gates (ports) *pin* and *pout*. A detailed discussion of the specification, including the full LOTOS source, can be found in [23].

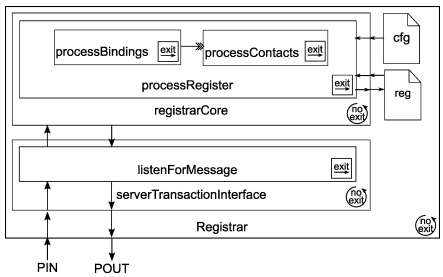

Figure 2. Main structure of the Registrar specification.

Table 1. Abstractions for the specification of the SIP Registrar.

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

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 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 [14], which are strategies from 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 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 encounter different test-design strategies. Among others, test cases are designed on basis of structural coverage criteria on the source code, for example, decision/condition coverage [14]. Another popular test case design strategy relies on fault models. Thus, test engineers or developers often anticipate defects relying on their domain knowledge, their intuition, or on errors previously made [2].

Conceptually, we thus propose to employ these two strategies for the design of reasonable test purposes. Thus, we rely on both, on test purposes from the specification's structural properties and on test purposes from the anticipated fault models. This two strategies, which have been 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 decision/condition (DC) coverage on our formal model. Thus, we designed our test purposes such that the generated test cases cover the LOTOS guards within the behavioral part of our specification in terms of DC coverage. In order to avoid test cases that detect the same errors, test purposes should be orthogonal, i.e., 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. Therefor the authors of [1] use mutation operators in order to generate faulty mutants from the original specification. They generate the IOLTS S_τ for the original specification and the IOLTS S_τ^M for every mutant M. Afterwards S_τ and S_τ^M 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, since this technique requires the construction of the complete IOLTS. For industrial-sized applications this raises severe scalability issues: For example, during the construction of the model for our SIP Registrar specification, the CADP toolbox 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 we are unable to translate large LOTOS models into IOLTSs, 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 the fault has been introduced in the mutant. Fortunately, we know where the specification has been mutated. Hence, the key idea is to mark the place of mutation in the LOTOS specification with additional labels (α, β). The slices can be calculated by using TGV and a special test purpose that only selects (accepts) α-labeled transitions and refuses β-labeled ones. The result of applying these *slicing-via-test-purpose* technique are two test processes (graphs), one for the original specification, and one for the mutation. Finally, in contrast to [1] the CADP-bisimulation check is done on the two test processes that reflect the relevant behaviour 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 *g2* and *g1* in Line 3 has been changed from *g2; g1;* (original) to *g1; g2;* (mutant). Both versions of the specification have been annotated with α and β. Note, that α and β are not in the alphabet 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 α, but refuses traces that contain β 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 *g1* and *g2*.

Now we hide α and β in the test graphs, i.e., we transform α and β to internal events (τ). Calculating the discriminating sequence (using CADP-Bisimulator) between the two test graphs leads to *g1; g2*. This is our new test purpose which is used on the original specification.

Formally, we generate a test purpose for a specification

```
1 process original[g1,g2,g3,α,β] : exit
2   g1; (
3     g2; ( g1; α; exit [] g2; exit )
4     []
5     β; g3; ( g1; exit [] g3; exit )
6   )
7 endproc
```

```
1 process mutant[g1,g2,g3,α,β] : exit
2   g1; (
3     g1; ( g2; α; exit [] g2; exit )
4     []
5     β; g3; ( g1; exit [] g3; exit )
6   )
7 endproc
```

Figure 3. Applying the event swap operator to a LOTOS process.

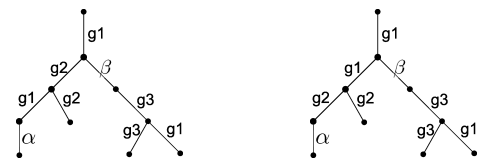

Figure 4. LTS representation of the original and the mutated specification.

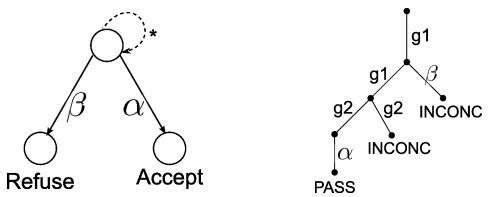

Figure 5. Test purpose for the extraction of marked traces and the resulting test graph.

$L = \left(Q^L, A^L, \rightarrow_L, q_0^L\right)$ 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 L'.
4. Generate two complete test graphs, CTG_τ for the specification and CTG_τ^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-Bcg). This leads to CTG'_τ = *hide* α, β *in* CTG_τ and to $CTG^{m\prime}_\tau$ = *hide* α, β *in* CTG^m_τ.

6. Minimize CTG'_τ and $CTG^{m\prime}_\tau$ using the Safety Equivalence relation in order to obtain CTG and CTG^m (using CADP-Reductor).

7. Check CTG and CTG^m for Strong Bisimulation (using CADP-Bisimulator). The counterexample c, if any, gives the new test purpose[1]. 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 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.

[1]The labels of the test processes are marked with *INPUT* or *OUTPUT*. We remove this marks.

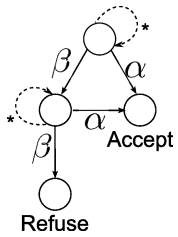

Figure 6. Extended test purpose for the extraction of marked traces.

6. Empirical Evaluation

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

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. A detailed discussion of our extraction algorithm can be found in [24]. 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 [24].

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. The behavioral part of the specification comprises 38 decisions with an average of 1.3 conditions per decision. At maximum there are three conditions in one decision.

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

Table 2. Test generation time.

Test purpose	no.tc	tgv [sec]	min [sec]	extr. [sec]
not found	880	12.7	0.4	13.7
interval too brief	384	12.3	0.6	4.5
invalid request	1328	12.7	1.0	19.3
unauthorized	660	24.1	0.7	9.7
register ok	1392	15.4	0.5	23.2
delete	2000	15404.6	0.7	25.9
Total	6644	15482.0	4.2	96.5

Table 3. Test execution results.

Test purpose	no.tc	commercial		OpenSER	
		pass	fail	pass	fail
not found	880	0	880	880	0
interval too brief	384	0	384	0	384
invalid request	1328	0	1328	1008	320
unauthorized	660	578	82	156	504
register ok	1392	1104	288	1104	288
delete	2000	18	1982	1439	561
Total	6644	1700	4944	4587	2057

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). The different test purposes address different parts of the specification, e.g., the test cases derived by the use of the *invalid request* test purpose cover the decisions/conditions of the specification that deal with invalid request messages.

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 outlines 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 proved detailed enough to reveal severe misbehavior and, as exemplified in terms of the OpenSER 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 Registrar we found 4 discrepancies between the IUT and the specification.

Most of the detected errors (commercial: 6, OpenSER: 2) are regarding different combinations of CONTACT header fields. For example, the commercial 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 requirement, because they delete stored contact information also if the CALL-ID numbers and the CSEQ values are equal.

Another detected error that causes many failed test cases 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 Registrar.

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 randomly 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. The table shows the number of possible mutants, 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 τ transitions (see Section 5.2.1). Note, that we encountered only one mutant for the event swap operator, where the fault is not visible 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 β. Since we run out of memory every time we allow 12 β before α, 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 CADP-Caesar 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. This 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.

Table 4. Test generation time.

Operator	No. Mutants				Avg. Time [sec]		
	poss.	appl.	ok	eqv.	ok	∞	tgv
eio	35	35	30	0	169	5696	3
eso	10	9	7	4	215	5564	5
mco	46	46	32	3	236	5374	1
Total	91	90	69	7	217	5465	2

Table 5. Test execution results.

Operator	no.	commercial			OpenSER		
	tc	pass	fail	inc.	pass	fail	inc.
eio	30	26	4	0	26	4	0
eso	3	2	1	0	3	0	0
mco	29	26	3	0	26	3	0
eio auth.	30	15	12	3	20	5	5
eso auth.	3	0	2	1	1	0	2
mco auth.	29	20	9	0	27	2	0
Total	124	89	31	4	103	14	7

7. Related Research

Various testing techniques have been applied to SIP [25, 18]. 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, 19, 20]. In difference to our formalization, the presented models are based on the outdated RFC 2543 [11]. 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 *ioco*-incorrect mutants. Kahlouche et al. present in [13] 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 hand-written 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. Unlike to other articles reporting on successful application of specification-based testing, we provide rationales for 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 second 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 SIP Registrar. 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 α and β.

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

[1] B. K. Aichernig and C. C. Delgado. From faults via test purposes to test cases: On the fault-based testing of concurrent systems. In *FASE*, volume 3922 of *LNCS*, pages 324–338. Springer, 2006.

[2] A. Beer and R. Rammler. Case studies on experience-based testing. Technical report, Software Competence Center Hagenberg, Siemens PSE, Competence Network Softnet Austria, To appear.

[3] S. Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119, IETF, 1997.

[4] K. Y. Chan and G. v. Bochmann. Modeling IETF session initiation protocol and its services in SDL. In *LNCS*, volume 2708, pages 352–373. Springer, 2003.

[5] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *TACAS*, volume 2280 of *LNCS*, pages 470–475. Springer, 2002.

[6] L. du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, and R. G. de Vries. Formal test automation: The conference protocol with TGV/TORX. In *TestCom*, volume 176 of *IFIP Conference Proceedings*, pages 221–228, 2000.

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

[8] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, 2002.

[9] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSC's. In *SDL'93, the 6th SDL Forum*, pages 253–266. Elsevier Science, 1993.

[10] W. Grieskamp, N. Tillmann, C. Campbell, W. Schulte, and M. Veanes. Action machines — towards a framework for model composition, exploration and conformance testing based on symbolic computation. In *QSIC*, pages 72–79, 2005.

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

[12] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.

[13] H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In *IWTCS*, pages 211–226, 1998.

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

[15] W. Prenninger and A. Pretschner. Abstractions for model-based testing. In *Proceedings Test and Analysis of Component-based Systems (TACoS'04)*, pages 59–71, 2004.

[16] 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 *ICSE*, pages 392 – 401, 2005.

[17] 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.

[18] H. Schulzrinne, S. Narayanan, J. Lennox, and M. Doyle. SIPstone - benchmarking SIP server performance. Technical report, Columbia University, Ubiquity, 2002.

[19] G. Stojsic, R. Radovic, and S. Srbljic. Formal definition of SIP end systems behavior. *EUROCON, Trends in Communications*, 2:293–296, 2001.

[20] G. Stojsic, R. Radovic, and S. Srbljic. Formal definition of SIP proxy behavior. *EUROCON Trends in Communications*, 2:289 – 292, 2001.

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

[22] J. Tretmans and E. Brinksma. Torx: Automated model based testing. In A. Hartman and K. Dussa-Zieger, editors, *Proceedings of the First European Conference on Model-Driven Software Engineering*, pages 13–25, Nurnburg, Germany, 2003.

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

[24] M. Weiglhofer. Conformance testing of a session initiation protocol server. Technical Report SNA-TR-2006-1P2, Competence Network Softnet Austria, To appear.

[25] C. Wieser, M. Laakso, and H. Schulzrinne. SIP robustness testing for large-scale use. In *SOQUA/TECOS*, volume 58 of *LNI*, pages 165–178, 2004.