ABSTRACT

Nowadays test engineers typically use two strategies for the design of test cases. First, test cases are designed related to some kind of structural coverage criteria. Second, test cases are created by having a specific fault model in mind. In this paper we evaluate these two supplementary techniques for test purpose design. We present a heuristic algorithm for the extraction of test cases from TGV’s output, i.e., the test process. We discuss the problem of overlapping test purposes and illustrate improvements in terms of test execution time and in terms of number of test cases when minimizing this overlap. Furthermore, we evaluate different strategies for the generation of fault-based test purposes on our specification. For our evaluation, all extracted test cases are executed against a commercial and an open source implementation of a Session Initiation Protocol (SIP) Registrar.

Categories and Subject Descriptors
D.2.5 [Testing and Debugging]: Testing tools;
D.2.4 [Software/Program Verification]: Formal methods

General Terms
Reliability, Experimentation

Keywords
model-based testing, protocol conformance testing, Session Initiation Protocol (SIP), fault-based test purposes, test purpose design

1. INTRODUCTION

Today’s ever increasing amount of software and software-enabled systems requires software systems to be well engineered and highly reliable. It is not only a product’s amount of software that makes software quality a central issue. Due to the ever increasing trend towards complex, distributed, and highly reactive systems, there is need to significantly reduce the fault detection and the fault removal costs given a desired level of quality. The complexity of both the software and the complexity of the testing itself yields to the desire for automation. Whereas automation of test execution in industrial scale software development is saliently addressed nowadays, the automation of test data generation is not that common. Although systematic test data generation allows for obtaining product-centric rather than process-centric quality metrics (and thus might improve the overall product quality).

Models are commonly used to master system complexity, and model-based testing (MBT) is concerned with deriving test cases from a model capturing the product’s intended behavior. For highly available and reactive systems this is often captured by the model of the underlying protocol. In these cases the central point is the conformance with respect to the protocol’s specification. In this article we report from an application of protocol conformance testing to a so called SIP Registrar in the context of a voice over IP (VoIP) server1. Due to the well-founded theory on input-output labeled transition systems (IOLTS), the ability of this theory to be employed in presence of non-deterministic models, and the availability of mature research prototypes, we decided to rely on IOLTSs.

This article is a companion paper to [2] and focuses on the comparison of structural and fault-based test purpose design. Unlike to [2] where we present the basic idea of fault-based test purpose design together with first empirical results, this article discusses our novel test case extraction

1We conducted the research reported in this article in collaboration with 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.
algorithm in detail, classifies our strategies for fault-based test purpose design, and presents new empirical results on the evaluation of so called overlapping test purposes. Both items were not included in [2]. While [2] extends fault-based test purpose design for specifications with huge state spaces, this paper contains detailed figures about different strategies for our fault-based test purpose design approach.

In this article we contribute to MBT particularly with respect to test purpose design. For structural test purpose design we propose a novel algorithm for extracting test cases from a given test graph, and identify so called overlapping test purposes. Empirical results on a commercial SIP Registrar and an open source implementation underpin that test purpose design is a noteworthy issue for industrial sized applicability of MBT. Moreover, we introduce novel strategies for fault purpose test design, and evaluate these strategies empirically. Our empirical results substantiate that particularly fault-based test purpose design offers further potential for leveraging MBT particularly under presence of large and complex specifications.

This paper continues as follows: in Section 2 we briefly introduce the input output conformance relation. In Section 3 we discuss structural test purpose design, overlapping test purposes and propose a novel test case extraction algorithm. Section 4 summarizes the ideas of fault-based test purpose design for specifications with huge state spaces and discusses different strategies for fault-based test purpose generation. In Section 5 we present empirical results and in Section 6 we discuss related work. Finally, in Section 7, we present our conclusions.

2. PRELIMINARIES

TGV [12] generates test cases in order to test input output conformance of an implementation with respect to an IOLTS model. TGV employs LOTOS as its primary input language, however, any other input language providing IOLTS semantics is applicable alike. We briefly review the input output conformance relation in the following.

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 [15].

**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 $A^{\text{in}} = A^{\text{in}} \cup A^{\text{out}} \cup \{\tau\}$ where $A^{\text{in}}$ and $A^{\text{out}}$ are input and output alphabets and $\tau \notin A^{\text{in}} \cup A^{\text{out}}$ 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 $q, q', q_1 \in Q^M$, $\tau \in \tau$, $a \in A^M$ and $\sigma \in \{A^{\text{in}} \cup A^{\text{out}}\}^*$. Then, $q \xrightarrow{a} q_1$ if $(q, a, q_1) \in \rightarrow_M$ and $q \xrightarrow{\tau} q'$ if $(q, a, q') \in \rightarrow_M$. If $\sigma \in \{A^{\text{in}} \cup A^{\text{out}}\}^*$ then $q \xrightarrow{\sigma} q'$. We denote $q \xrightarrow{d\tau} q'$ as $q \xrightarrow{\sigma} q'$ by $q \xrightarrow{\tau} q'$ which generalizes to $q \xrightarrow{\sigma \tau} q'$. We denote $q \xrightarrow{\alpha^M} \sigma = q' \cup \{q_1 \mid q \xrightarrow{\alpha} q_1 \}$ and $Out_M(Q) = \bigcup_{q \in Q} (Out_M(q))$. 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 $\Rightarrow$) when it is clear from the context.

Commonly the symbol $\delta$ is used to represent a quiescent state. 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. A LTS $M$ is called strongly responsive if it always eventually enters a quiescent state, that is $q \in Q^M \rightleftharpoons q \xrightarrow{\delta} q_{\text{after}}^M$. Note, that strongly responsive labelled transition systems do not have infinite loops labelled with the internal action $\tau$. We say, a LTS $M$ is strong input enabled if it accepts every input in every state: $\forall a \in A^M, \forall q \in Q^M : q \xrightarrow{a} \rightleftharpoons M$ is weak input enabled if it excepts either an internal action or all input actions in all states: $\forall a \in A^M, \forall q \in Q^M : q \xrightarrow{a} \rightleftharpoons \Rightarrow_{\text{weak}}$.

To define the input output conformance relation we need the set of suspension traces which is defined as $\text{Straces}(q) =_{\text{def}} \{ \sigma \in \{A^M \cup A^O \cup \{\delta\}\} \mid q \xrightarrow{\sigma} \}$.

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

**Definition 2.** The $\text{itoc}$ 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 $\text{IUT} = (Q^{\text{IUT}}, A^{\text{IUT}}, \rightarrow_{\text{IUT}}, q_0^{\text{IUT}})$ be weakly input enabled with $A^{\text{IUT}} = A^{\text{IUT}} \cup A^{\text{IUT}} \cup \{\tau\}$ and $S = (Q^S, A^S \cup \{\delta\}, q_0^S)$ be strongly responsive with $A^S \subseteq A^{\text{in}} \cup A^{\text{out}} \cup \{\tau\}$.

$IUT \text{ itoc} S =_{\text{def}} \forall \sigma \in \text{Straces}(S) :$

$$\text{Out}_{\text{IUT}}(\text{IUT \ after}_{\text{IUT}} \sigma) \subseteq \text{Out}_{\text{S \ after}}(\sigma)$$

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

2.2 Soundness and Exhaustiveness

The ideal test suite is complete, that is, it is sound and exhaustive. A sound test suite rejects non-conform implementations only. Thus it is guaranteed, that rejected implementations do not conform to the specification, i.e. there are no false negatives. An exhaustive test suite rejects all non-conform implementations. In case the implementation is faulty, an exhaustive test suite detects this, i.e there are no false positives. Formally, soundness, exhaustiveness, and completeness are defined as follows:

**Definition 3.** Given a specification $s$, any implementation $i$, and a test suite $T$, then

$T$ is sound =_{\text{def}} \forall i : i \text{ itoc } s \Rightarrow i \text{ passes } T$

$T$ is exhaustive =_{\text{def}} \forall i : i \text{ itoc } s \Leftarrow i \text{ passes } T$

$T$ is complete =_{\text{def}} \forall i : i \text{ itoc } s \Rightarrow i \text{ passes } T$

2.3 Test Purposes

While formal specifications 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. In conformance testing the notation of a test purpose has been standardized [10]:
TGV extracts the visible behavior of the synchronous product \(SP\) by adding suspension labels and applying determination to \(SP\), which leads to \(SP^{VIS}\). The determination removes internal actions \(\tau\) 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 by inverting outputs and inputs. States where an input is possible: \(\forall a \in A^{SP}\), \(q \in T C \Rightarrow \exists b \in A^{SP}\), \(q \leftarrow_{TC} a\). TGV may produce an exhaustive test suite.

**Theorem 1.** [12] For every specification \(S\), all test suites produced by TGV are sound. Moreover, the (infinite) test suite consisting of all test cases that TGV may produce is exhaustive.

**2.4 Test Case Extraction with TGV**

According to [12] test synthesis within TGV is conducted as follows. Given a test purpose TP and a specification S, TGV calculates the behavior of S accepted by TP. This is done by calculating the synchronous product \(SP = S \times TP\).

**Definition 6.** Let \(S = (Q^S, A^S, \rightarrow_S, q_0^S)\) be an IOLTS and \(TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})\) a test purpose with \(A^{TP} = A^S\) and equipped with state sets \(Accept^{TP}\) and \(Refuse^{TP}\). The synchronous product \(SP = S \times TP\) is an IOLTS \(SP = (Q^{SP}, A^{SP}, \rightarrow_{SP}, q_0^{SP})\), equipped with two disjoint sets of states \(Accept^{SP}\) and \(Refuse^{SP}\), and defined as follows:

- **Its alphabet** is \(A^{SP} \equiv \{A^{TP}\}\).
- **Its state set** \(Q^{SP}\) is the subset of \(Q^S \times Q^{TP}\) reachable from the initial state \(q_0^{SP} \equiv (q_0^S, q_0^{TP})\) by the transition relation \(\rightarrow_{SP}\).
- **The transition relation** \(\rightarrow_{SP}\) is defined by:
  \[\left(q^S, q^{TP}\right) \rightarrow_{SP} \left(q'^S, q'^{TP}\right) \iff q^S \rightarrow_S q'^S \land q^{TP} \rightarrow_{TP} q'^{TP}\].
- **Accept^{SP} and Refuse^{SP}** are defined as follows:
  \[Accept^{SP} \equiv Q^{SP} \cap \left(Q^S \times Accept^{TP}\right),\]
  \[Refuse^{SP} \equiv Q^{SP} \cap \left(Q^S \times Refuse^{TP}\right)\]

The derived test purposes suffer from one major problem. The test purposes overlap, i.e., the derived IOLTS test graphs have many common traces.

**Definition 7.** For a given specification \(s\), two test purposes \(tp_1, tp_2\) overlap if

\[Straces(tp_1 \times s) \cap Straces(tp_2 \times s) \neq \emptyset\]

Due to overlapping test purposes some of the generated test cases cover the same faults. In order to get rid of such common test cases we merge test purposes that select similar edges. For example, our specification allows many different stimuli. For many of these stimuli an implementation can accept the message by answering with an “OK” response. For some of them an implementation may respond with an “Interval too Brief” message. Thus, the two test purposes of Figure 1 can be merged to the test purpose illustrated in Figure 2.
3.1 Extracting Test Cases

TGV can either derive a test graph or single test case for a given test purpose and a certain specification. When using structural test purpose design we propose to extract test cases from the test graph instead of generating one test case for each test purpose. Especially in the case of data-dependent specifications, i.e. formal models which heavily use abstract data types, using the test graph allows the generation of many test cases from few test purposes.

Test graphs, generated by TGV, may contain loops, i.e., there may be infinitely many test cases. Thus, we need a heuristic in order to extract test cases from the test graph. Similar to the cycling heuristics from [4], we assume that each cycle needs to be traversed only a limited number of times. So our test cases iterate through every cycle in the test graph once. Furthermore we do not take all possible paths of the test graph, but only use transition coverage on the test graph to extract the test cases.

Our test case extraction algorithm is illustrated in Figure 3. The algorithm initializes the set of visited states \( v_s \) to the empty set, the count of how often an edge has been selected \( v_e \) to zero for any edge, and the count of equal test cases \( c_eq \) to zero (line 4). Because the algorithm should generate a test graph with transition coverage we generate test cases until every edge has been visited at least once (line 5). A single test case is generated by iterating through the test graph using a breath first search. The queue \( Q_{next} \) holds the states that should be processed next, which is initially \( q_0 \) (line 8). \( Q_{next} \) supports the 3 operations push, pop, which adds an element at the back end of the queue, pop_front, which retrieves the element at the front of the and removes the element from the queue, and empty which returns true if the queue does not contain any element and false otherwise.

The function getNextEdges selects the next edges for a given state \( p \) (line 11). Therefor the function chooses the edge, that has been selected at least often to be the next edge for the current test case (line 39). If the selected edge is labeled by an input label, we have to complete the set of edges by all other input labels (line 41). For every selected edge \( e = (q, a, q') \) the algorithm increases the selected count \( v_e \) by one (line 15). We add \( q' \) to the set of next states \( Q_{next} \) and we update the set of visited states \( v_s \), if the condition of line 16-17 holds for \( e \). The first part of the condition (line 16) ensures that we do not add verdict edges to \( Q_{next} \), because a verdict state is a sink state, i.e. it has no outgoing edges. The second part of the condition (line 17) ensures that we do not process any state twice, except states that are currently part of loops in the final test case.

Every selected edge \( e \) is added to the test case \( tc \), which is currently under generation (line 22). Note, that by adding the edge to the test case the algorithm has to take care, that the final test case does not contain loops.

If the generated test case \( tc \) is not yet in the set of generated test cases \( TC \), we add the test case to \( TC \). Otherwise we increase the count of equal generated test cases \( c_eq \). If \( c_eq \) is higher as a certain maximum, we abort the test case generation, in order to ensure that the algorithm terminates within reasonable time.

If the test graph is a valid test graph according to Section

### Figure 1: Two overlapping test purposes.

### Figure 2: Result of merging the test purposes of Figure 1.

### Figure 3: Heuristic test case extraction algorithm.
2.5, then all test cases derived from the test graph by this algorithm are valid test cases. That is, the derived test cases fulfill the additional property of controllability. Since we add all input edges at a certain state, when an input edge is selected, the test case is input complete if the test graph is input complete in states where inputs are possible.

If this algorithm is applied to a sound test process (graph), the resulting set of test cases is sound too. This is because the derived test cases preserve input completeness from the test graph, the derived test cases preserve the verdict states from the test graph, and the derived test cases only contain traces that are in the test graph.

However, because of the used heuristic our test cases are not exhaustive. Since we assume, that one iteration through a loop is enough, we loose the exhaustiveness property of the test graph from TGV. For example, our test cases are not able to detect faults that occur in the second iteration of a loop.

4. FAULT-BASED TEST PURPOSE DESIGN

Our second technique for test purpose design is based on fault models. Its primary idea is to prevent the implementation under test to conform to a faulty specification. [1] outlines details regarding fault-based test purpose design. Briefly we summarize the basic principle:

Mutation operators are used to generate faulty mutants from the original specification. The original specification and the mutants are transformed to their IOLTS representations under test to conform to a faulty specification. [1]

4.1 Coping with Large Specifications

To overcome these intricacies, we propose to use only the parts of the specification which exhibit the introduced fault [2]. Fortunately, we know where the specification has been mutated. Hence, the key idea is to mark the place of mutation in the LTS representation 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. In contrast to [1], the CADP-Bisimulation check is done on the two test processes that reflect the relevant behavior of their models. Since we perform the equivalence check on the test processes rather than on the models itself, we overcome the scalability issues regarding the model’s size. The following example of [2] illustrates this technique.

Figure 4 shows 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 language of the original specification L, i.e., \{α, β\} ⊈ A^L. The labeled transition systems described by the specification and the mutant are depicted in Figure 5.

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 6 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 the ordering of g1 and g2.

Now we hide α and β in the test graphs, i.e., we transform α and β to the internal event 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 L = (Q^L, A^L, →L, q^L_0) 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 \{α, β\} \nin A^L into the formal (LOTOS) specification L.
3. Generate a mutated version of the specification L” = O_m(L’) by applying O_m to the marked, formal specification L’.
4. Generate two complete test graphs, \( CTG_\alpha \) for the specification and \( CTG_{\alpha m} \) for the mutant, by the use of the test purpose from Figure 6 (using CADP-TGV).

5. Hide the additional added labels by transforming them to internal transitions \( \tau \) (using CADP-Bcg). This leads to \( CTG_\alpha = \text{hide } \alpha, \beta \text{ in } CTG_\alpha \) and to \( CTG_{\alpha m} = \text{hide } \alpha, \beta \text{ in } CTG_{\alpha m} \).

6. Minimize \( CTG_\alpha \) and \( CTG_{\alpha m} \) using the Safety Equivalence relation in order to obtain \( CTG \) and \( CTG_{\alpha m} \) (using CADP-Reducer).

7. Check \( CTG \) and \( CTG_{\alpha m} \) for Strong Bisimulation (using CADP-Bisimulator). The counterexample \( c \), if any, gives the new test purpose\(^2\). \( c \) 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 \( \alpha \) cannot be reached from \( q_\alpha^{\alpha m} \). In that case any sequence \( c \) of \( CTG \) that ends in \( \alpha \) is a discriminating sequence. By hiding \( \alpha \) and \( \beta \) in \( c \) and possibly adding one more valid transition, we obtain our new test purpose.

### 4.2 Marking Strategies

The capability of the outlined approach depends on the insertion strategy of the markers \( \alpha \) and \( \beta \). While we use different insertion strategies for \( \beta \), we insert \( \alpha \) subsequently to the position of the scheduled mutation. For example, when using the event swap operator, as indicated in Figure 4, we insert \( \alpha \) after the swapped events.

In the test purpose \( \beta \) leads to the Refuse-state. Thus for construction the test graph, we employ \( \beta \) to prune the search space by cutting individual paths. If there is no \( \beta \)-free path from the initial state of the IOLTS to \( \alpha \), \( TGV \) is unable to generate a test graph since there exists no path leading to an Accept-state. Hence, the insertion strategy for \( \beta \) has to take care about the reachability of \( \alpha \).

In the following we introduce three novel strategies for inserting \( \beta \), namely the Process-Local strategy, the Breath First Search strategy, and the Depth First Search strategy. We afterward outline our novel empirical results obtained from the individual strategies.

\(^2\)The labels of the test processes are marked with INPUT or OUTPUT. We remove this marks.

### 5. EMPIRICAL EVALUATION
For an empirical evaluation we applied structural and fault-based test purpose design to the specification of an entity of the Session Initiation Protocol (SIP), namely the SIP Registrar. We executed the generated test cases against a commercial implementation and against OpenSER\(^3\), an open source SIP Registrar.

5.1 SIP Registrar Specification

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 message 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 9. 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-sent the register request. The second request is encrypted with the HTTP-Digest method [6]. This request is accepted by the Registrar and answered with "200 OK". For a full description of SIP we refer to [11].

In cooperation with our 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 (contribution 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. Details about our SIP Registrar specification can be found in [16].

5.2 Test execution

No matter which test purpose design we might rely on, the obtained test cases are abstract ones. The transitions of an abstract test case capture stimuli and the implementation’s expected responses in an abstract manner. For test execution, these abstract 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 [17].

In order to ensure a particular system state of the implementation under test for each test case we reset the implementation before running a certain test case. For our empirical evaluation, the test execution procedures reside on the same computer as the implementation under test in order to ensure that messages are delivered reliable and in the sent order.

We overcome the problems of concurrency and asynchronous communication by using the reasonable environment assumption [5] which says that, each time the environment sends a message to the network, it waits until stabilization. This means, that the test execution environment is not allowed to send new messages until it received all responses from the implementation. This assumption avoids the crossing of test messages. Since, we only have a single bidirectional point of control and observation, and because of the particular structure of our specification, this assumption solves the problem of asynchronous communication as well.

5.3 Results for Structural Test Purposes

Table 1 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).

Table 2 uses the structure of Table 1 to illustrate the figures of test generation with minimized overlapping between
the test purposes. When optimizing the test purposes with respect to their overlapping the amount of time needed to generate the test graphs decreases significantly. Due to the reduction of the overlapping, the number of generated test cases decreases as well. This implies a reduction of the time needed for executing the test cases.

However, the coverage of the test suites on the source code of OpenSER (we obtained 78% function coverage and 36% condition/decision coverage according to Bullseye Coverage Tool\(^4\)), remains unchanged. With respect to the performed abstractions [2] the obtained test cases cover the majority of the code implementing the SIP Registrar’s functionality. The uncovered part of code mainly deals with error handling (e.g. database update errors), with saving contacts in memory only (a feature that we turned off for our experiments), and with functions for retrieving stored contact information. This code is not subject to test in terms of our formal model.

Although, the reduction of the overlap between our test purposes, does not change the behavior of TGV on the delete test purpose. For this test purpose we observe, that TGV needs more time for the generation of the test graph, than for our other test purposes. The delete 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. However, the rewriting of our test purposes bisects the generation time for the delete test purpose.

Table 3 and Table 4 outline the results of executing the obtained test cases from our overlapping and from non-overlapping test purposes against the commercial and the OpenSER Registrar in terms of the number of executed (2nd column), passed (3rd and 6th column), failed test cases (4th and 7th column) and the time needed to execute the test cases (5th and 8th column). This time includes the amount of time needed to reset the implementation, which is approximately 2.7 sec. per test case. If the implementation under test does not respond to a request, we need to wait for the expiration of a timer (3 sec.) during test execution. This may cause different test execution durations on different implementations. For example, the execution of the not found test suite consumes much more time when executed on the commercial SIP Registrar than on the open source SIP Registrar. This is because the commercial SIP Registrar does not respond to the test messages of this test suite.

Remarkably, we are able to reduce the execution time by approximately 13% for the commercial SIP Registrar and by approximately 5% for the OpenSER SIP Registrar.

Due to our specification’s structure, we obtain various similar test cases. In summary, we detected diverse faults: 9 different faults for the commercial implementation and 4 distinct faults for the OpenSER implementation. For example, our formal model rejects every register request for an unknown user. Since there are many possible register request we derive many test cases that check if registrations for unknown users are rejected correctly. Because the rejection does not depend on any other field than the user we have many test cases testing the same feature. In [2] we discuss the various types of errors detected in detail.

### 5.4 Results for Fault-Based Test Purposes

Due to the size of our 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 proposed in [1]. However, we chose three mutation operators (event insert operator (eio), event swap operator (eso), missing condition operator (mco)) and applied our different strategies manually.

Table 5, 6, and 7 list details about our three novel strategies for fault-based test generation. These tables show 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 \( \tau \) transitions (see Section 4.1). The 4th column shows how many of the generated test graphs are equivalent, i.e., there is no observable difference between the mutant and

---

\(^4\)http://www.bullseye.com
Table 5: Test generation results for fault-based test purpose generation using the process-local marking strategy.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pos. appl.</td>
<td>eqv.</td>
<td>TP</td>
<td>ok</td>
<td>∞</td>
</tr>
<tr>
<td>eso</td>
<td>10</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>eio</td>
<td>35</td>
<td>35</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>mco</td>
<td>46</td>
<td>46</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>Total</td>
<td>91</td>
<td>90</td>
<td>3</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 6: Test generation results for fault-based test purpose generation using the breath first search strategy.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pos. appl.</td>
<td>eqv.</td>
<td>TP</td>
<td>ok</td>
<td>∞</td>
</tr>
<tr>
<td>eso</td>
<td>10</td>
<td>9</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>eio</td>
<td>35</td>
<td>35</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>mco</td>
<td>46</td>
<td>46</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>Total</td>
<td>91</td>
<td>90</td>
<td>7</td>
<td>69</td>
</tr>
</tbody>
</table>

Figure 10: Cumulative number of generated test purposes for 0-10 unrolled β-steps.

Figure 11 shows the evolvement of the number of test purposes when allowing additional β-steps before α. The majority of test cases (50) are already generated by unrolling 20 β.

Comparison. In difference to the other two strategies, the process-local strategy has the advantage, that we only need to run TGV once. We do not need to unroll dependencies by the use of the test purpose. Thus, the average times for test purpose construction are in both cases, the case of success (101 seconds) or the case of out of memory (3490 seconds), lower than for the other two strategies (217 sec, 1642 sec and 5465 sec, 9009 sec respectively).

Remarkably, the depth first search strategy does not run out of memory. Even after allowing 60 β-steps before α the memory consumption stays below 2 GB. Thus, it may be possible to generate more mutants by unrolling more than 60 β.

However, as illustrated in Figure 12, the breath first search strategy outperforms the other two approaches. Note, that this figure uses log-scale for the x-axis. Using the breath first search we are able to generate 69 test purposes within approximately 535 seconds. In difference to that, the depth first search strategy needs 9485 seconds to generate 64 test purposes. Specifically for our specification, the process-local marking strategy generates 12 test purposes only. This evaluation shows, that if the structure of the specification allows the insertion of many β for the process-local strategy, the process-local strategy should be used. In other cases, the breath first search strategy should be applied.

Figure 10: Cumulative number of generated test purposes for 0-10 unrolled β-steps.

Figure 11 shows the evolvement of the number of test purposes when allowing additional β-steps before α. The majority of test cases (50) are already generated by unrolling 20 β.

Comparison. In difference to the other two strategies, the process-local strategy has the advantage, that we only need to run TGV once. We do not need to unroll dependencies by the use of the test purpose. Thus, the average times for test purpose construction are in both cases, the case of success (101 seconds) or the case of out of memory (3490 seconds), lower than for the other two strategies (217 sec, 1642 sec and 5465 sec, 9009 sec respectively).

Remarkably, the depth first search strategy does not run out of memory. Even after allowing 60 β-steps before α the memory consumption stays below 2 GB. Thus, it may be possible to generate more mutants by unrolling more than 60 β.

However, as illustrated in Figure 12, the breath first search strategy outperforms the other two approaches. Note, that this figure uses log-scale for the x-axis. Using the breath first search we are able to generate 69 test purposes within approximately 535 seconds. In difference to that, the depth first search strategy needs 9485 seconds to generate 64 test purposes. Specifically for our specification, the process-local marking strategy generates 12 test purposes only. This evaluation shows, that if the structure of the specification allows the insertion of many β for the process-local strategy, the process-local strategy should be used. In other cases, the breath first search strategy should be applied.

Table 5: Test generation results for fault-based test purpose generation using the process-local marking strategy.

Table 6: Test generation results for fault-based test purpose generation using the breath first search strategy.

Table 7 illustrates the results for fault-based test purpose generation using the depth first search strategy. We are able to generate 7, 27, and 30 test purposes out of 9, 35, and 46 mutants within an average time of 1642 seconds. 7 of the created mutants are equivalent to the original specification. For the other 26 mutants, TGV was not able to generate a test graph even after allowing 60 β-steps before α.

The production times are also included, while the 7th β-step before α.

Figure 10 shows the evolvement of the number of test purposes when allowing additional β-steps before α. The majority of test cases (50) are already generated by unrolling 20 β.

Comparison. In difference to the other two strategies, the process-local strategy has the advantage, that we only need to run TGV once. We do not need to unroll dependencies by the use of the test purpose. Thus, the average times for test purpose construction are in both cases, the case of success (101 seconds) or the case of out of memory (3490 seconds), lower than for the other two strategies (217 sec, 1642 sec and 5465 sec, 9009 sec respectively).

Remarkably, the depth first search strategy does not run out of memory. Even after allowing 60 β-steps before α the memory consumption stays below 2 GB. Thus, it may be possible to generate more mutants by unrolling more than 60 β.

However, as illustrated in Figure 12, the breath first search strategy outperforms the other two approaches. Note, that this figure uses log-scale for the x-axis. Using the breath first search we are able to generate 69 test purposes within approximately 535 seconds. In difference to that, the depth first search strategy needs 9485 seconds to generate 64 test purposes. Specifically for our specification, the process-local marking strategy generates 12 test purposes only. This evaluation shows, that if the structure of the specification allows the insertion of many β for the process-local strategy, the process-local strategy should be used. In other cases, the breath first search strategy should be applied.
The test cases were evaluated on 24 poses and by the use of randomly-generated test purposes. These were generated by the use of manually-designed test purposes. Even after ten hours of test purpose design, the authors did not manage to come up with a set of test purposes that detects all faulty mutants. This case study illustrates the complexity of test purpose design in detail. Furthermore, these case studies do not consider mutation testing.

### 6. RELATED RESEARCH

There exist various case studies on using TGV for automatic test generation. For example, the authors of [3] 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 isoco-incorrect mutants. This case study illustrates the complexity of test purpose design. Even after ten hours of test purpose design, the authors did not manage to come up with a set of test purposes that detects all faulty mutants.

Kahlouche et al. present in [13] the application of TGV to the cache coherency protocol, while the authors of [5] 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. Furthermore, these case studies do not consider mutation testing.

### 7. CONCLUSION

In this article we contribute to MBT particularly with respect to test purpose generation. For structural test purpose design we propose a novel algorithm for extracting test cases from a given test graph and identify so called overlapping test purposes. Empirical results on a commercial SIP Registrar and an open source implementation underpin that test purpose design is a noteworthy issue to further advance model-based testing (MBT).

Moreover we introduce novel strategies for fault test purpose design: The process-local, the breath first search, and the depth first search strategy. In terms of empirically evaluating our strategies on the so called SIP Registrar, we provide a first assessment of our novel approaches: If the specification is structured appropriately, the process-local strategy appears to be beneficial. In the case of our SIP Registrar specification, the breath first search strategy outperforms the other two approaches in terms of running time and in terms of generated test purposes. For this specification, the breath first search strategy is approximately an order of magnitude faster than the depth first search strategy. It allows to generate 5 and 45 more test purposes than the depth first search strategy and the process-local strategy, respectively.

However, our approach needs further evaluation. We need to automate the generation of mutants for LOTOS specifications in order to evaluate our different strategies for all
mutation operators. For test cases generated from structural designed test purposes the ratio between the number of failing test cases and the number of detected faults is yet very high, so there is still some potential for optimizations. While this work identifies the problem of overlapping test purposes and the effects of reducing this overlap, we need general rules for rewriting test purposes. This needs further investigation.

8. REFERENCES


During the review, the submitted version is available at http://www.ist.tugraz.at/staff/weiglhofer/sefm2007.pdf.


