Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 220 (2008) 63–77

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Improving Fault-based Conformance Testing[1]

Bernhard K. Aichernig[2] Martin Weiglhofer[3] Franz Wotawa[4]

Institute for Software Technology
Graz University of Technology
Graz, Austria

Abstract

Fault-based conformance testing is a conformance testing strategy that relies on specific fault models. Previously, this mutation testing technique has been applied to protocol specifications. Although a practical case study of web-server testing has been conducted, we observed several issues when applying this method in a large industrial project. In this paper, we discuss the foundations, techniques and tools to overcome these shortcomings. More specifically, we show a solution to the problem of state-space explosion in generating mutation tests for industrial scale applications. Furthermore, the previous approach used the counterexamples of a bisimulation check (between the original and the mutant) as test purposes. With respect to input-output conformance (ioco), this is an over-approximation resulting in more tests than are necessary. Hence, we propose to use an ioco-checker in order to generate less test cases. An industrial case study demonstrates these improvements.

Keywords: mutation testing, labelled transition systems, input-output conformance, ioco, CADP-TGV, Session Initiation Protocol (SIP)

1 Introduction

Nowadays software and software-enabled systems are becoming increasingly complex. Model-based testing techniques assist in systematic testing of such systems. By starting from a formal model test cases are derived automatically in order to test the conformance of implementations with respect to their specification.

However, given a formal specification there is a huge, possibly infinite, number of test cases that can be derived from that specification. There are different ways of selecting a finite set of test cases. One possibility is the use of coverage criteria on the level of the specification for test case selection. Another way, is the use of anticipated faults for the generation of test cases. This approach of test case selection, which is subject to this paper, is also known as mutation testing and was

[1] Authors are listed in alphabetical order.

[2] Email: aichernig@ist.tugraz.at

[3] Email: weiglhofer@ist.tugraz.at

[4] Email: wotawa@ist.tugraz.at

doi:10.1016/j.entcs.2008.11.006

introduced in [7]. Basically, a fault is modeled at the specification level by altering the specification. The idea is to generate test cases that would find such faults in implementations.

One example of such a technique has been presented in [1], where the authors generate test cases by the use of test purposes derived from mutated specifications. They use mutation operators in order to generate faulty mutants from the original specification. Afterwards, the labeled transition systems for the original specification and for the mutant are constructed and minimized to their observable behavior using the safety equivalence reduction of CADP [10]. A bisimulation check on the minimized labeled transition systems gives a discriminating sequence if there is an observable difference between the mutant and the original. This sequence serves as test purpose for the TGV tool [13]. The generated test case fails on implementations that conform to the mutated specification.

This approach has several advantages. First, it provides a reasonable test selection strategy by focusing on faults on the specification level. Second, this approach is applicable to non-deterministic systems, since the derived test cases have a tree-like structure.

Although, [1] showed the practicability of this approach by testing a web-server, we observed several open issues when applying the method in an industrial project. First, the used bisimulation check over-approximates the set of needed test cases, because the generated test cases are intended for input-output conformance (ioco) testing with respect to the ioco-relation of Tretmans [20]. Albeit, due to the use of TGV all generated test cases are sound, i.e. test cases do not fail on input-output conform implementations. However, there are test cases for faults that can not be detected using the ioco-relation. Thus, using bisimulation results into more test cases than needed. Second, the original approach relies on the construction of the complete state spaces of both, the original specification and the mutated specification. For industrial specifications with huge state spaces this is often infeasible. In this paper we address these two problems.

This paper continues as follows. In Section 2 we review the underlying testing theory. In Section 3 we develop the foundations for a new ioco-based test case selection approach. Section 4 shows how to deal with large specifications and Section 5 outlines the overall approach. Section 6 shows our experimental results. In Section 7 we briefly review related work. Finally, we conclude in Section 8.

2 Preliminaries

Our approach relies on the ioco testing theory of [20]. Thus, we briefly review the ioco relation. For a detailed discussion of ioco we refer to [20].

2.1 Input-Output Conformance

In this section we introduce the models for test case generation that are used to describe specifications, implementations, test cases and test purposes.

Definition 2.1 An input output labeled transition system (IOLTS) is a labeled transition system $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ 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 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 labeled transition systems 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 $q \xrightarrow{a}_M =_{df} \exists q' : (q, a, q') \in \rightarrow_M$. $q \overset{\epsilon}{\Rightarrow} q' =_{df} ((q = q') \vee (q \xrightarrow{\tau}_M q_1 \wedge \cdots \wedge q_{n-1} \xrightarrow{\tau}_M q'))$ and $q \overset{a}{\Rightarrow} q' =_{df} \exists q_1, q_2 : q \overset{\epsilon}{\Rightarrow}_M q_1 \xrightarrow{a}_M q_2 \overset{\epsilon}{\Rightarrow}_M q'$ which generalizes to $q \overset{\sigma}{\Rightarrow} q' = q \overset{a_1 \ldots a_n}{\Rightarrow} q' =_{df} \exists q_0, \ldots, q_n : q = q_0 \overset{a_1}{\Rightarrow}_M q_1 \ldots q_{n-1} \overset{a_n}{\Rightarrow}_M q_n = q'$. We denote $q\ \mathbf{after}_M\ \sigma =_{df} \{q' |\ q \overset{\sigma}{\Rightarrow}_M q'\}$ and $Q\ \mathbf{after}_M\ \sigma =_{df} \bigcup_{q \in Q}(q\ \mathbf{after}_M\ \sigma)$. We define $Out_M(q) =_{df} \{a \in A_O^M |\ q \xrightarrow{a}_M\}$ and $Out_M(Q) =_{df} \bigcup_{q \in Q}(Out_M(q))$. $Traces(M)$ denotes all possible sequences of actions $\sigma \in (A_I^M \cup A_O^M)^*$. We will omit the subscript $_M$ (and superscript ^{M}) when it is clear from the context.

An IOLTS M is *weakly input enabled* if it accepts all inputs in all states, possibly after internal τ actions: $\forall a \in A_I^M, \forall q \in Q^M : q \overset{a}{\Longrightarrow}$. An IOLTS is *deterministic* if for any trace there is at most one successor state, i.e., $\forall \sigma \in \left(A_I^M \cup A_O^M\right)^* : |q_0^M\ \mathbf{after}_M \sigma| \leq 1$, where $|X|$ denotes the cardinality of the set X. An IOLTS is *complete* if it allows all actions in each state, i.e. $\forall q \in Q^M, \forall a \in A^M : q \xrightarrow{a}_M$.

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 unobservable action. Thus, $q \xrightarrow{\delta} q$ means, that q is a quiescent state. To define the ioco relation we need the suspension automaton, which makes quiescence observable by considering δ as an output.

Definition 2.2 *The suspension automaton of an IOLTS* $S = \left(Q^S, A^S, \rightarrow_S, q_0^S\right)$ *is an IOLTS* $\Delta(S) = (Q^S, A^{\Delta(S)}, \rightarrow_{\Delta(S)}, q_0^S)$ *where* $A^{\Delta(S)} = A^S \cup \{\delta\}$ *with* $\delta \in A_O^{\Delta(S)}$. *The transition relation* $\rightarrow_{\Delta(S)}$ *is obtained from* $\rightarrow_S$ *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. The following definition of the ioco relation says, that an implementation I conforms to a specification S, iff the outputs of I are outputs of S after an arbitrary suspension trace of S.

Definition 2.3 *Let* S *be an IOLTS and* I *be an weakly input enabled IOLTS, where the alphabets of* I *and* S *are compatible, i.e.,* $A_I^S \subseteq A_I^I$, *and* $A_O^S \subseteq A_O^I$, *then*

$$I\ \boldsymbol{ioco}\ S =_{df} \forall \sigma \in Straces(S) : Out_I(\Delta(I)\ \boldsymbol{after}\ \sigma) \subseteq Out_S(\Delta(S)\ \boldsymbol{after}\ \sigma).$$

2.2 Test Purposes and Test Synthesis with TGV

While a formal model is a description of the system under test, a test purpose can be seen as a formal specification of a test case. Tools like TGV [13] use test purposes for test case generation. TGV defines a test purposes as follows [13]:

Definition 2.4 *A test purpose is a complete, deterministic IOLTS* $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$*, equipped with two sets of trap states* $Accept^{TP}$ *and* $Refuse^{TP}$*, with the same alphabet as the specification S, i.e.* $A^{TP} = A^S$*. A trap state q has a self-loop for each action, i.e.* $\forall a \in A^{TP} : q \xrightarrow{a}_{TP} q$*.*

The states in $Accept^{TP}$ of a test purpose define pass verdicts, while states in $Refuse^{TP}$ allow to limit the exploration of the specification's state space. That is, $Refuse^{TP}$ states are not explored during test case generation. More precisely, according to [13] test synthesis with TGV is conducted as follows. Given a test purpose TP and a specification S TGV calculates the synchronous product $SP = S \times TP$. Then TGV extracts the visible behavior SP^{VIS} of SP by adding suspension labels and applying determinization to SP. The determinization removes τ actions 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 Cases, Test Graphs and Test Suites

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

Definition 2.5 *A test case is a deterministic IOLTS* $TC = \left(Q^{TC}, A^{TC}, \rightarrow_{TC}, q_0^{TC}\right)$ *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:*

(i) *TC mirrors image of actions and considers all possible outputs of the IUT:* $A^{TC} = A_I^{TC} \cup A_O^{TC}$ *with* $A_I^{TC} \supseteq A_O^{IUT} \cup \{\delta\}$ *and* $A_O^{TC} \subseteq A_I^{IUT}$*.*

(ii) *From each state a verdict must be reachable:* $\forall q \in Q^{TC}, \exists \sigma \in A^{TC*}, \exists q' \in Pass \cup Inconc \cup Fail : q \overset{\sigma}{\Rightarrow}_{TC} q'$*.*

(iii) *States in* $Fail$ *and* $Inconc$ *are only directly reachable by inputs:* $\forall (q, a, q') \in \rightarrow_{TC}$: $(q' \in Inconc \cup Fail \Rightarrow a \in A_I^{TC})$*.*

(iv) *A test case is input complete in all states where an input is possible:* $\forall q \in Q^{TC} : (\exists a \in A_I^{TC}, q \xrightarrow{a}_{TC} \Rightarrow \forall b \in A_I^{TC}, q \xrightarrow{b}_{TC})$*.*

(v) *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 a test graph already satisfies the properties of a test case. A *test suite* is a set of test cases.

3 A IOCO-Approach to Fault-based Testing

Basically, we use mutation operators at the specification level in order to generate faulty versions of a specification. The mutation of the specification affects the underlying formal model, which is an input-output labeled transition system in our case. Not all mutations represent faults. A fault can only be defined with respect to a conformance relation. In the following we show the meaning of faults in the context of ioco.

3.1 From Faults via Non-conformance to Test Cases

Our first observation is that not all injected faults will cause observable failures. In order to observe a failure, the mutant must not be io-conform to our original specification. Hence, given an original specification S we are only interested in mutants S^m, such that $\neg\ (S^m\ \mathbf{ioco}\ S)$[5]. Unfolding the definition of **ioco** gives

$$
\begin{aligned}
&\neg\ (S^m\ \mathbf{ioco}\ S) = && (1)\\
&= \neg\ (\forall \sigma \in STraces(S) : Out(\Delta(S^m)\ \mathbf{after}\ \sigma) \subseteq Out(\Delta(S)\ \mathbf{after}\ \sigma)) && (2)\\
&= \exists \sigma \in STraces(S) : \neg(Out(\Delta(S^m)\ \mathbf{after}\ \sigma) \subseteq Out(\Delta(S)\ \mathbf{after}\ \sigma)) && (3)\\
&= \exists \sigma \in STraces(S) : Out(\Delta(S^m)\ \mathbf{after}\ \sigma) \not\subseteq Out(\Delta(S)\ \mathbf{after}\ \sigma)) && (4)\\
&= \exists \sigma \in STraces(S) : \\
&\qquad \exists o : o \in Out(\Delta(S^m)\ \mathbf{after}\ \sigma) \wedge o \notin Out(\Delta(S)\ \mathbf{after}\ \sigma)) && (5)
\end{aligned}
$$

Line 3 is the first hint for a testing strategy. We are interested in the suspension trace of actions leading to non-conformance between the mutant and the original IOLTS. In other words, our test purposes for detecting faults are sequences of actions leading to non-conformance. We immediately derive the fact that such a non-conformance checking can be reduced to a test for subsets on the output labels (Line 4). It follows the fact (Line 5) that a failure is observed, if the mutant S^m produces an output o not predicted by specification S. This simple derivation shows an important property of the ioco conformance relation: mutating the specification by injecting an additional inputs a such that a new trace for the mutant S^m is generated, i.e. $\forall \sigma \in STraces(S) : \sigma \cdot a \notin STraces(S)$, does not lead to a failure.

The theory highlights a further important clarification in fault-based testing: In presence of non-determinism, there is no guarantee that an actual fault will always be detected. The reason is that non-conformance only means that there is a wrong output after a trace of actions, but the implementation may still opt for the correct one. In that case we rely on the complete testing assumption [14], which says that an implementation exercises all possible execution paths of a test case t, when t is applied a finite number of times.

However, for some faults a strengthened condition, i.e. all outputs are wrong with respect to the specification, may hold. Formally, that is

$$\exists \sigma \in STraces(S) : \forall\ o \in Out(\Delta(S^m)\ \mathbf{after}\ \sigma) : o \notin Out(\Delta(S)\ \mathbf{after}\ \sigma)$$

[5] We make S^m input enabled.

In that case we need to run the test case only once, because if this specific fault has been implemented there is no right output the implementation can opt for.

In the future we should label the test cases that aim to detect faults which may be hidden due to non-determinism. Such test cases need to be executed more than once in order to raise the confidence in the implementation's conformance. On the other hand, we can identify test cases that require only a single execution.

4 State Space Reduction Techniques

In this section we show how to apply fault-based conformance testing to specifications with huge state spaces. Given a mutant and a specification we need a discriminating sequence that exhibits the difference between the mutant and the specification. The idea is to extract a slice from the specification that includes the relevant parts only instead of comparing the complete state spaces. The relevant part for our conformance check are the places where the fault has been introduced into 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 S with additional labels (α, β) that are not used in the initial alphabet of S.

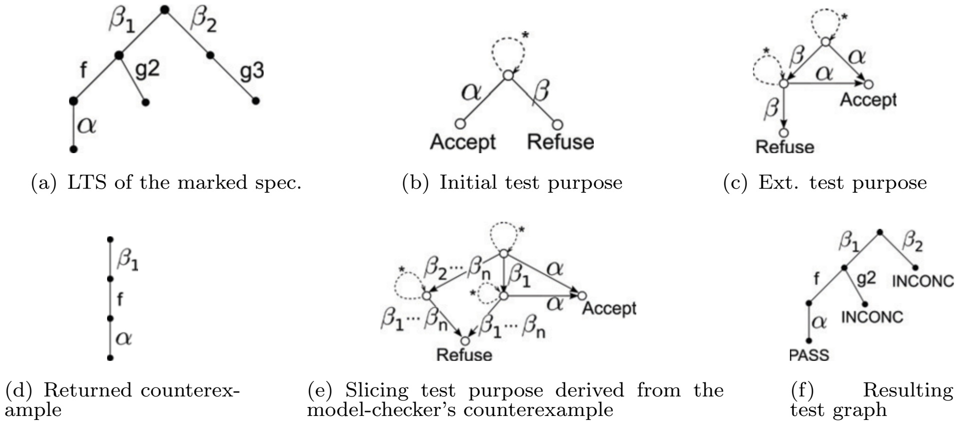
Basically, after a LOTOS event leading to the fault injected by the mutation, an α event is inserted. Furthermore, we insert a β event before every other event of the LOTOS specification. Then, the slices can be calculated using TGV and a special test purpose that only selects (accepts) α-labeled transitions and refuses β-labeled ones. The result of applying this *slicing-via-test-purpose* technique are two test processes (graphs), one for the original specification, and one for the mutant. Finally, the discriminating sequence is extracted from the two test processes that reflect the relevant behavior of their models. Hence, the size of the model does not matter any more since the equivalence check is performed on the test processes.

Note, that we rely on LOTOS [12] specifications for our discussion. Since LOTOS does not distinguish between input and output events, i.e. LOTOS has an LTS semantics, input-output information is added during test case generation. However, our approach is applicable to any other specification language with (IO)LTS semantics.

4.1 Incremental Slicing

This idea has been presented in [3]. However, the capabilities of this approach depend on the search strategy of the α marker. We used an incremental strategy. In the first step we search for a β-free path to an α. If such a path does not exist our slicing strategy (our test purpose) was too strong. Thus, we have to allow more βs before α in the *slicing-test-purpose*.

For example, if we use the test purpose of Fig. 1(b) to slice the LTS of Fig. 1(a) TGV fails to generate a test graph. Thus, the test purpose is extended to the one depicted in Fig. 1(c), permitting one β transition in the path to α. For this example TGV is able generate a test graph using this extended test purpose. However, if TGV fails again to produce a test graph (using the new test purpose), we have to repeat



(a) LTS of the marked spec.

(b) Initial test purpose

(c) Ext. test purpose

(d) Returned counterexample

(e) Slicing test purpose derived from the model-checker's counterexample

(f) Resulting test graph

Fig. 1. LTSs and test purposes illustrating the *slicing-via-test-purpose* technique.

this procedure n times until a test graph can be generated.

This *breadth-first-search* approach has two major drawbacks. First, we need to invoke TGV once, for every β that is in the path leading to α, which results into many unnecessary calls of TGV. Second, by enabling a single β we may select more transitions of the labeled transition system than needed. For example, using the extended test purpose of Fig. 1(c) on the LTS of Fig. 1(a) results into the whole LTS since TGV never encounters a refuse state. To overcome this defiances we propose to use an on-the-fly model-checker.

4.2 Reachability Analysis with Model Checkers

Basically, we need a trace that ends in α and contains all βs marking the events leading to α. By using an on-the-fly model-checker and a safety property, i.e. a property that informally states that "something bad never happens", we can generate such a trace. If we state that α is not reachable, a model-checker's counterexample is a trace that ends in α. Since, we previously inserted α such a trace always exists if the specification contains no unreachable parts.

The counterexample returned by the model-checker is a trace that contains all βs which should be allowed before α. If we make the β events distinguishable, e.g. by adding a unique number to each β event, the counterexample reflects the β event that should be allowed by the *slicing test purpose*.

For example, running the model-checker on the labeled transition system depicted in Fig. 1(a) results in the counterexample shown in Fig. 1(d). This counterexample says that we should allow the event β_1 in order to reach α. Translating this counterexample results into the test purpose illustrated in Fig. 1(e). If we apply this test purpose to the labeled transition system of Fig. 1(a), the result looks like the LTS depicted in Fig. 1(f).

Care has to be taken that parts of the faulty behavior are not sliced away. Hence, an important precondition for this technique is that the insertion of the β actions does not interfere with the effect of the mutation. That is, if the mutation does not produce an equivalent mutant, the labeled transition system has to exhibit the injected fault between a certain β action and the α action. Otherwise, the constructed test purpose may cut the part of the specification which exhibits the injected fault.

Due to our simple α, β insertion strategy we currently cannot apply our approach to all mutation operators of [1]. Basically, we are limited to mutations that do not effect the tail state of a transition, i.e. mutations where the fault is exhibited by the transition represented by the mutated LOTOS event. For example, the process operator replacement (POR) mutation operator replaces process instantiations with stop and exit events. Since, we do not know which actions are effected by this mutation we cannot insert an α action after the mutation. Note, that smarter marking strategies, e.g. strategies that make use of a specification's control flow dependencies, would allow to overcome these limitations.

By using a model-checker we only need to invoke TGV once for the calculation of the relevant part. Furthermore, due to the distinguishable βs the test graph reflects the relevant part of the specification more precisely.

5 The Overall Approach

Using the findings of Section 3 and the state space reduction technique of Section 4 we can now reformulate the test purpose generation procedure of [1]. Thus, we generate a test purpose for a specification $L = \left(Q^L, A^L, \rightarrow_L, q_0^L\right)$ as follows:

(i) Select a mutation operator O_m.

(ii) Use the knowledge where O_m changes the specification to generate L' by inserting markers $\{\alpha, \beta\} \not\subseteq A^L$ into the formal specification L.

(iii) Generate a mutated version L^m of the specification L' by applying O_m

(iv) Call the model-checker to generate a trace that ends in α (using CADP-evaluator [15]) and derive a *slicing-test-purpose* from this trace.

(v) Generate two complete test graphs, CTG_τ for the specification and CTG_τ^m for the mutant, by the use of the *slicing-test-purpose* (using TGV).

(vi) Transform CTG_τ and CTG_τ^m to CTG_τ' and $CTG_\tau^{m\prime}$ by hiding the marker labels α and β (using CADP-Bcg [10]).

(vii) Check CTG and CTG^m for input output conformance (using an ioco checker [22]). The counterexample c, if any, gives the new test purpose[6].

6 Testing the Session Initiation Protocol

This section discusses our experimental results when applying our approach to the Session Initiation Protocol Registrar. We conducted all our experiments on a PC with AMD Athlon(tm) 64 X2 Dual Core Processor 4200+ and 2GB RAM.

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

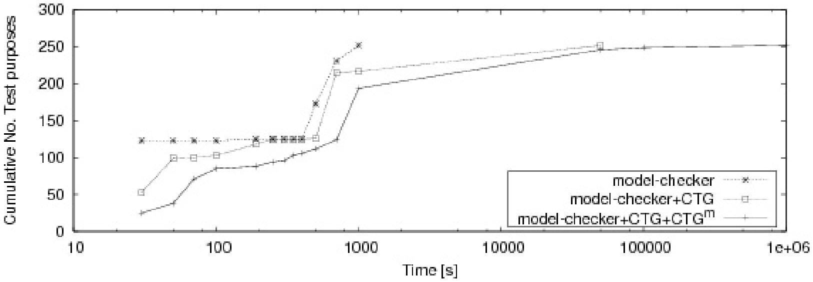


Fig. 2. Time needed by the different steps in our test purpose generation process.

6.1 Session Initiation Protocol Registrar

The Session Initiation Protocol (SIP) [19] 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* (i.e., availability and location of end systems) and for *session management* (i.e. establishment of sessions, session modifications, . . .). One entity within a SIP network is the so called Registrar, which is responsible for the user management.

In cooperation with our industry partner's domain experts we developed a formal specification covering the functionality of a Registrar. This LOTOS specification consists of approx. 3KLOC (net.), 20 data types (net. 2.5KLOC), and 10 processes. Details about our specification can be found in [21].

6.2 Test Case Generation Results

We developed a mutation tool that takes a LOTOS specification and uses the applicable mutation operators (see Section 4.2) in order to generate for each possible mutation one faulty version (mutant) of the specification. Additionally, our mutation tool inserts the α and β markers according to the strategy of Section 4.2. By the use of this mutation tool we generated the mutants of our specification.

Table 1 lists for each mutation operator (1st and 2nd column) the overall number of possible mutants (3rd column) and the number of mutants for which our marking strategy of Section 4 is applicable (4th column).

In addition, Table 1 shows the average time needed for the particular test case generation steps. The 5th column lists the average number of seconds needed to extract a path that leads to the mutation using the CADP on-the-fly model-checker [15]. Finally, this table depicts the average number of seconds needed by TGV [13] to extract the relevant parts of the original specification S (6th column) and of the mutated specification S^m (7th column).

As this table shows, the average time needed to generate the counterexample using the model-checker is 283 seconds. Additionally TGV needs on average 680 seconds to generate the test graph for the original specification and 9072 seconds for the generation of the mutant's test graph. Fig. 2 explains these high average numbers, showing that the sum of time needed for the calculation of the different

oper-	description	no.	no.	avg.	avg.	avg.
ator		mut.	marked	mc	tgv S	tgv S^m
ASO	Associative Shift Op.	11	5	402	142	145
CRO	Channel Replacement Op.	37	37	260	723	8257
EDO	Event Drop Op.	31	31	404	2102	1982
EIO	Event Insert Op.	35	35	128	1383	1296
ENO	Expression Negation Op.	52	3	183	65	68084
ERO	Event Replacement Op.	35	35	373	878	254
ESO	Event Swap Op.	9	9	83	2116	1785
LRO	Logical Operators Replacement	21	8	340	117	31981
MCO	Missing Condition Op.	46	11	208	105	907
ORO	Operand Replacement Op.	416	32	258	98	97
RRO	Relational Operators Replacement	91	12	218	85	1411
SNO	Simple Expression Negation Op.	32	7	435	152	912
USO	Unobservable Sequence Op.	27	27	388	879	818
total		**843**	**252**	**283**	**680**	**9072**

Table 1
Number of generated mutants and timing results for the extraction of the relevant parts for the original specification and the mutated specification.

results during the test purpose generation process over the cumulative number of generated test purposes. For example, the line for `model-checker+CTG+CTGm` shows the sum of time needed to run the model-checker, to calculate the complete test graphs CTG and CTG^m for the specification and for the mutant using TGV.

Up to 77% of our 252 test purposes can be generated within reasonable time (i.e. 1000 seconds per test case). The other 23% test purposes are regarding complex scenarios of our specification (e.g., client registers and removes its registrations afterwards). These complex scenarios push the average up.

Table 2 illustrates the difference between using a bisimulation check as proposed in [1] and using a ioco check as discussed in Section 3. This table shows for every mutation operator (1st column), the number of mutants (2nd column), the average time needed to compare the extracted test processes (3rd and 7th column), the number of equivalent mutants (4th and 8th column) and the number of mutants with an observable difference (5th and 9th column) when using bisimulation (3rd-6th column) and when using input-output conformance (7th-10th column). Furthermore, this table shows the average amount of time (in seconds) needed for generating the final test case using TGV (6th and 10th column). Note, we did not investigate in the runtime performance of our ioco checker yet.

As Table 2 illustrates, the total number of generated test purposes, and thus the total number of generated test cases decreases by using the input-output conformance relation. Instead of using an over-approximated set of test cases comprising 154 test cases, we only need to execute 98 test cases.

Basically, the ioco-equivalent mutants come from mutations where input actions have been affected. For example, in our specification we use guards on inputs in order to avoid meaningless input actions. If a mutation operator weakens such a condition, e.g. by removing parts of the condition (MCO), the mutant allows more

[7] The equality of the rows four and nine and of the rows five and eight is a coincidence.

op.	no.	bisimulation			tc gen-	ioco			tc gen-
	mark.	time [sec]	=	≠	eration	time [sec]	=	≠	eration
ASO	5	3.43	5	0	-	7.07	5	0	-
CRO	37	3.39	7	30	2.18	101.35	9	28	2.34
EDO	31	3.87	17	14	4.19	131.51	17	14	4.39
EIO	35	3.84	10	25	4.26	316.33	13	22	4.59
ENO	3	4.59	1	2	-	365.74	1	2	-
ERO	35	2.87	16	19	2.59	79.76	16	19	2.92
ESO	9	4.43	4	5	3.98	8.04	4	5	3.79
LRO	8	2.96	2	6	1.68	216.51	8	0	-
MCO	11	3.91	4	7	2.68	187.50	11	0	-
ORO	32	5.61	8	24	3.54	13.46	32	0	-
RRO	12	6.94	3	9	3.86	180.56	12	0	-
SNO	7	6.31	2	5	2.65	32.17	7	0	-
USO	27	4.37	19	8	3.06	155.16	19	8	2.51
total[7]	**252**	**4.35**	**98**	**154**	**3.15**	**138.01**	**154**	**98**	**3.42**

Table 2
Bisimulation check, ioco check, and test case generation results.

inputs than the original specification. However, when testing with respect to the input-output conformance relation injecting new additional inputs does not lead to failures, as shown in Section 3.

The incremental slicing technique, which has been evaluated in [4], failed on some complex mutants of our specification. More precisely, incremental slicing failed on approximately 14% of the evaluated mutants. In difference to that, using the CADP on-the-fly model-checker allows to generate sliced state spaces for all of our mutations.

6.3 Test Execution Results

We executed our test cases against the open source implementation OpenSER and against two versions (V1, V2) of a commercial SIP Registrar implementation.

Table 3 illustrates the number of test cases (2nd column), the number of passed (columns 3, 6, and 9), failed (columns 4, 7, and 10) and inconclusive (columns 5, 8, and 11) test runs when using the bisimulation relation (3rd-18th row) and when using the ioco relation (19th-34th row) for every mutation operator (1st column). From the 4th to the 16th row and from the 20th to the 32nd row we list the results for running the test cases against the Registrars with authentication turned on, while the 17th row and the 33rd row list the results with authentication turned off.

By the use of the generated test cases we detected 3 different faults in the open source, 5 different faults on V1 and 4 different faults on V2 of the commercial implementation. However, the verdict fail does not imply that the corresponding mutant has been implemented. It also happens that there occurred an failure during the execution of the preamble of the test case. The preamble is the sequence of messages that aims to bring the implementation to a certain state in which the difference between the mutant and the original specification can be observed. Furthermore, a test case's verdict is inconclusive if the implementation chooses outputs different to the outputs required by the test case's preamble. That is, the chosen output

op.	no.	OpenSER			v1			v2		
	tc	✓	✗	?	✓	✗	?	✓	✗	?
bisimulation relation										
CRO	30	3	26	1	5	23	2	5	23	2
EDO	14	8	3	3	0	1	13	0	1	13
EIO	25	14	3	8	7	0	18	7	0	18
ENO	2	0	2	0	0	2	0	0	2	0
ERO	19	2	17	0	2	13	4	2	13	4
ESO	5	0	5	0	0	4	1	0	4	1
LRO	6	0	6	0	0	5	1	0	5	1
MCO	7	0	7	0	0	5	2	0	5	2
ORO	24	0	24	0	0	16	8	0	16	8
RRO	9	0	9	0	0	6	3	0	6	3
SNO	5	0	5	0	0	5	0	0	5	0
USO	8	0	8	0	0	6	2	0	6	2
subtotal	154	27	115	12	14	86	54	14	86	54
wo.auth	154	0	103	51	0	108	46	0	93	61
total	**308**	**27**	**218**	**63**	**14**	**194**	**100**	**14**	**179**	**115**
ioco relation										
CRO	28	2	24	2	3	23	2	3	23	2
EDO	14	9	1	4	1	1	12	1	2	11
EIO	22	13	3	6	6	0	16	6	0	16
ENO	2	0	2	0	0	2	0	0	2	0
ERO	19	2	16	1	3	13	3	3	13	3
ESO	5	0	5	0	0	4	1	0	4	1
LRO	0		-			-			-	
MCO	0		-			-			-	
ORO	0		-			-			-	
RRO	0		-			-			-	
SNO	0		-			-			-	
USO	8	1	6	1	1	6	1	1	6	1
subtotal	98	27	57	14	14	49	35	14	50	34
wo.auth	98	4	82	12	0	89	9	0	68	30
total	**196**	**31**	**139**	**26**	**14**	**138**	**44**	**14**	**118**	**64**

Table 3
Results when execution the generated test cases on different SIP Registrars.

is correct with respect to the specification, but the test case failed to bring the implementation to the required state.

The tests generated on basis of the ioco relation detect the same faults as the test cases generated on basis of the bisimulation relation. Hence, the 56 additional test cases of the bisimulation approach did not reveal additional faults. Consequently, these test cases fail because of faults unrelated to their mutants.

7 Related Work

In extension to [3] we evaluated in [4] different *slicing-via-test-purpose* strategies, which rely on incremental slicing. None of them was able to generate test purposes for all mutants, in contrast to the approach presented in this paper.

Test purpose generation has been subject to previous research [11,8]. The authors of [8] present a modified model-checking algorithm that allows to transform properties, given in computational tree logic (CTL), to test purposes. Henniger et al. [11] automatically generate test purposes by identifying significant behavior of a system. Each signification behavior is converted to a test purpose. However, both articles do not consider testing for specific faults.

Ammann et al. [5] suggested the use of model-checkers for mutation based test case generation. While model-checker based mutation testing was limited to deterministic models recent research [6,16] allows the application of model-checkers to non-deterministic models. These approaches allow the use of non-deterministic specification. However, the result of mutation based model-checker test case generation is still a linear trace that serves as test case. Applying such a test case to non-deterministic implementations (or to implementations where the response to a stimuli does not only depend on the test case's input) may cause incorrect verdicts.

The authors of [9] showed how to use linear test cases for testing non-deterministic systems. States with possible non-deterministic choices are marked with inconclusive verdicts. If an inconclusive verdict is derived an additional call of the model-checker verifies the correctness of the response with respect to the model. Although, this makes linear test cases applicable to non-deterministic systems, test execution may become very slow due to many calls of the model-checker during test execution.

Petrenko and Yevtushenko [18] showed how to use partial, non-deterministic finite state machines (FSM) for mutation based test case generation. This work makes FSM based testing more amenable in industrial applications where specifications are rarely deterministic and complete. The difference to our approach is the used model. FSMs assume that a system cannot accept a next input before producing an output as a reaction to a previous input.

It is out of scope of this paper to review the growing related work on fault-based testing on the specification level. For an extensive discussion on related work in this area we refer to [2].

8 Conclusion

In this paper, we have developed a new ioco-based test case generation technique focusing on faults. It has been shown that this represents an advancement to previous approaches [1,3], from both a theoretical and an experimental point of view: (1) In contrast to the previous approach we generate ioco-relevant test cases only. In the case of our Session Initiation Protocol (SIP) Registrar experiments this means a reduction of 36% in the overall number of test cases. (2) Furthermore, we pushed the limits of this technique by developing an improved slicing technique. In the case of our SIP Registrar experiments the incremental slicing technique runs out of memory for complex scenarios which did not happen for the presented model-checker based slicing technique. We strongly believe, that these advancements make fault-based conformance testing more amenable to industrial-scale testing.

Relying on the TGV tool, the main limits of our approach stem from the wellknown assumptions of the input-output conformance (ioco) relation. For pros, cons and alternatives see for example [17].

The main issues to be addressed in the future are: First we should optimize the mutation operators using the findings of Section 3 in order to reduce the number of equivalent mutants. Second, our slicing method limits the applicable mutation operators. We need to develop a more general method that allows the application of all mutation operators. Third, while this paper only gives empirical evidence of the feasibility of our approach a formal correctness argument needs to be developed.

Acknowledgement

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). Additionally, this research is partially funded by the EU project, Framework 6: IST-33826 CREDO.

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

References

[1] Aichernig, B. K. and C. C. Delgado, *From faults via test purposes to test cases: On the fault-based testing of concurrent systems.*, in: *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, 2006, pp. 324–338.

[2] Aichernig, B. K. and J. He, *Mutation testing in UTP*, Formal Aspects of Computing (2008), accepted for publication.

[3] Aichernig, B. K., B. Peischl, M. Weiglhofer and F. Wotawa, *Protocol conformance testing a SIP registrar: an industrial application of formal methods*, in: *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods* (2007), pp. 215–224.

[4] Aichernig, B. K., B. Peischl, M. Weiglhofer and F. Wotawa, *Test purpose generation in an industrial application*, in: *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, 2007, pp. 115–125.

[5] Ammann, P., P. Black and W. Majurski, *Using model checking to generate tests from specifications*, in: *Proceedings of the Second International Conference on Formal Engineering Methods*, 1998, pp. 46–54.

[6] Boroday, S., A. Petrenko and R. Groz, *Can a model checker generate tests for non-deterministic systems?*, Electronic Notes in Theoretical Computer Science **190** (2007), pp. 3–19.

[7] Budd, T. A. and A. S. Gopal, *Program testing by specification mutation*, Computer languages **10** (1985), pp. 63–73.

[8] da Silva, D. A. and P. D. L. Machado, *Towards test purpose generation from CTL properties for reactive systems*, Electronic Notes in Theoretical Computer Science **164** (2006), pp. 29–40.

[9] Fraser, G. and F. Wotawa, *Nondeterministic testing with linear model-checker counterexamples*, in: *Seventh International Conference on Quality Software*, 2007, pp. 107–116.

[10] Garavel, H., F. Lang and R. Mateescu, *An overview of CADP 2001*, EAST Newsletter **4** (2002), pp. 13–24.

[11] Henniger, O., M. Lu and H. Ural, *Automatic generation of test purposes for testing distributed systems*, in: *3rd International Workshop on Formal Approaches to Testing of Software* (2003), pp. 178–191.

[12] ISO, *ISO 8807: LOTOS – a formal description technique based on the temporal ordering of observational behaviour* (1989).

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

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

[15] Mateescu, R. and M. Sighireanu, *Efficient on-the-fly model-checking for regular alternation-free mu-calculus*, Science of Computer Programming – Special issue on formal methods for industrial critical systems **46** (2000), pp. 255–281.

[16] Okun, V., P. E. Black and Y. Yesha, *Testing with model checker: Insuring fault visibility*, in: *Proceedings of the International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering System*, 2002, pp. 1351–1356.

[17] Petrenko, A. and N. Yevtushenko, *Queued testing of transition systems with inputs and outputs*, in: R. Hierons and T. Jéron, editors, *Proceedings of the Workshop Formal Approaches to Testing of Software*, 2002, pp. 79–93.

[18] Petrenko, A. and N. Yevtushenko, *Conformance tests as checking experiments for partial nondeterministic FSM*, in: *Proceedings of the 5th International Workshop on Formal Approaches to Software Testing*, LNCS **3997** (2005), pp. 118–133.

[19] Rosenberg, J., H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley and E. Schooler, *SIP: Session initiation protocol*, RFC 3261 (2002).

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

[21] Weiglhofer, M., *A LOTOS formalization of SIP*, Technical Report SNA-TR-2006-1P1, Competence Network Softnet Austria (2006).

[22] Weiglhofer, M. and F. Wotawa, *"On the fly" input output conformance verification*, in: *Proceedings of the IASTED International Conference on Software Engineering*, 2008, pp. 286–291.