From Faults Via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems

Bernhard K. Aichernig and Carlo Corrales Delgado

International Institute for Software Technology,
United Nations University, Macau SAR China,
P.O. Box 3058, Macau
{bka, carlo}@iist.unu.edu

Abstract. Fault-based testing is a technique where testers anticipate errors in a system under test in order to assess or generate test cases. The idea is to have enough test cases capable of detecting these anticipated errors. This paper presents a theory and technique for generating fault-based test cases for concurrent systems. The novel idea is to generate test purposes from faults that have been injected into a model of the system under test. Such test purposes form a specification of a more detailed test case that can detect the injected fault. The theory is based on the notion of refinement. The technique is automated using the TGV test case generator and an equivalence checker of the CADP tools. A case study of testing web servers demonstrates the practicability of the approach.

1 Introduction

The area of specification-based testing has advanced over the last couple of years and the results contributed to a reconciliation between the testing and the formal verification communities. Testing is now commonly acknowledged as a complementary V&V technique, if carried out systematically and well-founded. Gaudel was the most prominent who started this process [1]. Since then, many techniques and tools have emerged that generate test cases from formal specifications and are based on complete and sound testing theories.

However, the field is far from being complete, as the growing number of publications in this area indicates. Non-classical testing paradigms have to be studied and incorporated into theories. The formal underpinnings allow a deeper understanding of the relationships between testing and other verification theories, like simulation and refinement. This will lead to further applications of tools, like model checkers and constraint solvers, to generate test cases.

In this paper, we present a method that aims to advance the field in the following directions. (1) Mutation testing, traditionally applied to program text is applied on the specification level. (2) A model checker is not used to generate the test cases directly, but to generate test purposes, a high-level description of the testing goal. The method is founded on the testing theories on labeled transition systems. Tool support comes from the TGV test case generator [2] as well as from the CADP tools [3]. In particular, we address the following problems.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 324–338, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Problem 1 **Lack of test selection strategy.**

The early work on conformance testing in the area of distributed systems was mainly concerned with the soundness and completeness of the testing theory. Emphasis was given to develop a realistic conformance relation and a test case generation algorithm that was sound (no false negatives) and complete (no false positives). Since the models were finite labeled transition systems (LTSs), the problem of how to select a manageable subset out of the exhaustive test set was not a major concern. Abstraction was used to cope with the complexity. This lack of a test selection strategy limited the application domain to highly abstract protocol specifications.

Problem 2 **Identifying test purposes.**

To overcome this shortcoming, test purposes have been introduced. Here, a test purpose is a special LTS that specifies the subset of test cases to be generated. With test purposes, a tester can steer a test case generator according to his strategy. However, the problem remains, how many and which test purposes to select. Thus, the problem has been lifted, but not entirely solved.

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

Problem 3 **Equivalent mutants.**

A common problem in this approach is known as the Equivalent Mutant Problem. Not all mutations represent actual faults that can be observed at the interface level. Thus, no test case exists that can distinguish the original from such an equivalent mutant.

On the specification level, equivalence checkers can be used to eliminate such equivalent mutants. The problem is which equivalence relation, based on simulations, is appropriate for our purposes. Once, the equivalence relation is fixed, the problem is solved.

Problem 4 **Test generation automation.**

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

Therefore, we propose to use the counterexample as a test purpose. A test case generator, then, will generate a proper test case to cover the counterexample. Hence, our idea is to generate test purposes from injected faults, such that the

generated test cases will discover this anticipated faults. Fault-prevention, not structural coverage is our testing strategy.

The paper is organized as follows. After this introduction, Section 2 introduces the models, the testing theory as well as the concept of test purpose. Then, Section 3 develops the general properties of fault-based test purposes. Next, Section 4 presents the technique to generate test purposes using the CADP tools. A case study, briefly summarized in Section 5 completes the picture. Finally, we draw the conclusions and discuss related work in Section 6.

2 Conformance Testing

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. These models are based on the classical formalism of labelled transition systems (LTSs) with distinguished inputs and outputs. For a full definition of the testing theory we refer to [4].

2.1 Input-Output Conformance

Definition 1 (Input-Output LTS). *An IOLTS is an 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* $A^M = A_I^M \cup A_O^M \cup I^M$ *where* A_I^M *and* A_O^M *are respectively input and output alphabets and* I^M *is an alphabet of unobservable, internal actions,* $\rightarrow_M \subset Q^M \times A^M \times Q^M$ *is the transition relation and* q_0^M *is the initial state.*

We will use the following classical notations of LTSs for IOLTSs. Let $q, q', q_{(i)} \in Q^M, Q \subseteq Q^M, a_{(i)} \in A_I^M \cup A_O^M, \tau_{(i)} \in I^M$, and $\sigma \in (A_I^M \cup A_O^M)^*$. $q \stackrel{\epsilon}{\Rightarrow} q' =_{df} (q = q' \vee q \stackrel{\tau_1 \ldots \tau_n}{\rightarrow}_M q')$ and $q \stackrel{a}{\Rightarrow} q' =_{df} \exists q_1, q_2 : q \stackrel{\epsilon}{\Rightarrow}_M q_1 \stackrel{a}{\rightarrow}_M q_2 \stackrel{\epsilon}{\Rightarrow}_M q'$ which generalizes to $q \stackrel{a_1 \ldots a_n}{\Rightarrow} q' =_{df} \exists q_0, \ldots, q_n : q = q_0 \stackrel{a_1}{\Rightarrow}_M q_1 \ldots q_{n-1} \stackrel{a}{\rightarrow}_M q_n = q'$. We denote q **after**$_M$ $\sigma =_{df} \{q' \mid q \stackrel{\sigma}{\Rightarrow}_M q'\}$ and Q **after**$_M$ $\sigma =_{df} \bigcup_{q \in Q} (q$ **after**$_M$ $\sigma)$. We define $Out_M(q) =_{df} \{a \in A_O^M \mid q \stackrel{a}{\rightarrow}_M\}$ and $Out_M(Q) =_{df} \{Out_M(q) | q \in Q\}$. 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.

A specification is given in a formal description language which semantics allows to describe the behavior of the specification by an IOLTS (e.g. CSP, Estelle, SDL or LOTOS). The testing assumption is that the behavior of the implementation under test (IUT) can also be described by an IOLTS which can never refuse an input.

Definition 2 (Conformance). *The conformance relation says that an IUT conforms to S iff after a trace of S, outputs of the IUT are outputs of S:*

$$IUT \textbf{ ioconf } S =_{df} \forall \sigma \in Trace(S) : Out(IUT \textbf{ after}_{IUT} \sigma) \subseteq Out(S \textbf{ after}_S \sigma)$$

Note that this is a simplified version of **ioco** [4] excluding quiescence for the sake of clarity. All results apply to **ioco** as well.

2.2 Test Purposes

Test cases for complex concurrent systems correspond to elaborate executable programs: testing a certain procedure may require (1) to initialize a set of test processes that collaborate, (2) to execute a given preamble before being able to call the procedure, (3) to run an oracle process giving a verdict if the test has passed, and (4) to execute a postamble to get the system under test into a safe state after the test has been performed. In the telecom industry, TTCN [5] a special language for expressing test cases is used. It includes classical elements of programming languages: data types, variables, control structures and procedures.

In order to cope with the complexity of real test cases, test purposes serve to specify the goals of a test. Hence, a test purpose is a specification of a test case capturing the essence of a test in a short and abstract description. In conformance testing the notion of test purpose has been standardized [6]:

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

For generating test cases from test purposes, the notion of test purpose has been formalized, and implemented in tools like SAMSTAG [7], TGV [2], TorX [8], and most recently in Microsoft's XRT [9]. Here, we use the formalization of TGV.

Definition 4 (Test purpose, formal). *Given a specification S in form of an IOLTS, a test purpose is a deterministic IOLTS $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ equipped with two sets of sink states, $Accept^{TP}$ which defines Pass verdicts and $Refuse^{TP}$ which allows to limit the exploration of the graph S. Furthermore, $A^{TP} = A^S$ and TP is complete ($\forall q \in Q^{TP}, a \in A^{TP} : q \xrightarrow{a}_{TP}$).*

The specification to be covered by the test cases is formed by the synchronous product of the specification S and the test purpose TP. Furthermore, as test generation only considers the observable behavior of S it can be simplified by replacing all internal actions by τ, reducing the τ actions, and determinizing the result. This reduced specification SP_{VIS} is equipped with $Accept^{VIS}$ and $Refuse^{VIS}$ sink states derived from the test purpose.

2.3 Test Cases

In this testing framework for concurrent systems, a test case is a process running in parallel to the IUT. Hence, test cases can be modeled as an IOLTS that synchronize with the model of the IUT. TGV generates such test cases from the specification and a test purpose according to the algorithm described in [2].

Here, we only give the properties of a test case $TC = (Q^{TC}, A^{TC}, \rightarrow_{TC}, q_0^{TC})$ that has been generated from the restricted and simplified specification $SP_{VIS} = (Q^{VIS}, A^{VIS}, \rightarrow_{VIS}, q_0^{VIS})$ containing only visible actions. A test case TC has the following properties:

1. $Q^{TC} \subset Q^{VIS} \cup \{fail\}$, and $q_0^{TC} = q_0^{VIS}$,
2. $A^{TC} = A_I^{TC} \cup A_O^{TC}$ with $A_I^{TC} \subseteq A_O^{IUT}$ and $A_O^{TC} \subseteq A_I^{VIS}$ (mirror image of actions and all possible outputs of IUT considered),
3. $Pass = Accept^{VIS} \cap Q^{TC}$, $Inconc \subseteq Q^{VIS}$ and $fail$ are sink states and every state of TC except $fail$ can reach either a $Pass$ or an $Inconc$ state, $fail$ and $Inconc$ states can be reached directly only by inputs,
4. $\forall q \in Q^{TC}, \forall a \in A_I^{TC} : (\exists q' \in Inconc \cup \{fail\} : q \xrightarrow{a}_{TC} q' \Rightarrow q \xrightarrow{*} Pass)$ and $(q \xrightarrow{a}_{TC} fail \Rightarrow q \not\xrightarrow{a}_{VIS})$,
5. $\forall q \in Inconc : q \not\xrightarrow{*}_{VIS} Accept$ (Accept states cannot be reached from inconclusive states),
6. $\forall q \in Q^{TC}, \forall a \in A_O^{TC} : q \xrightarrow{a}_{TC} \Rightarrow \ \forall b \neq a : q \not\xrightarrow{b}_{TC}$ (only one output action per state).

For illustration purposes, Figure 1 shows the model of a coffee machine, with a test purpose and a resulting test case. Note that other test cases would be possible for this test purpose.

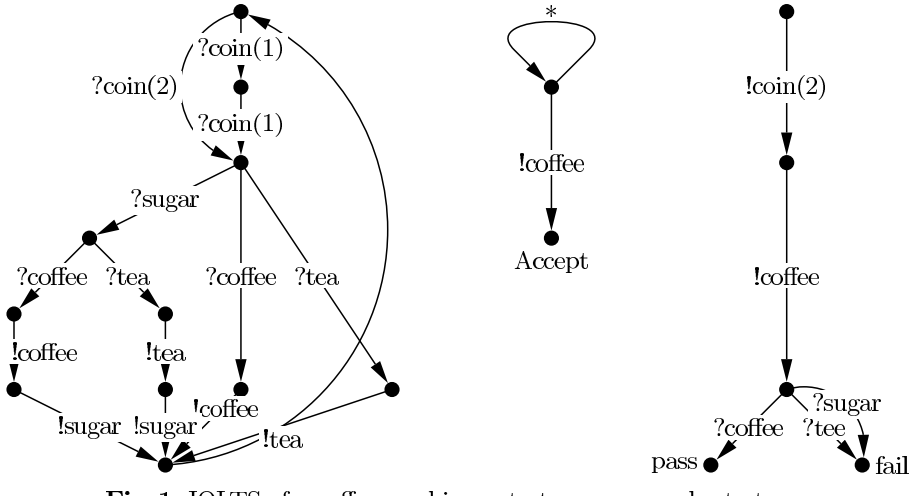

Fig. 1. IOLTS of a coffee machine, a test purpose, and a test case

3 A Theory of Fault-Based Testing

In this section, we develop a general fault-based testing theory that incorporates test purposes. As in our previous work, we use the concept of refinement as the basis to define what kind of test cases we are interested in. The difference here

is that we use the notion of refinement from [10] which relates test cases, test purposes and specifications.

3.1 Fault-Based Testing

Fault-based testing was born in practice when testers started to assess the adequacy of their test cases by first injecting faults into their programs, and then by observing if the test cases could detect these faults. This technique of mutating the source code became well-known as mutation testing and goes back to the late 70-ies [11, 12]; since then it has found many applications and has become the major assessment technique in empirical studies on new test case selection techniques [13]. In the early 90-ies formal methods entered the testing stage [14, 15] and it took not long until mutation testing was applied to formal specification languages [16]. Here, the idea is to model design errors or misinterpretations of requirements in a very early stage, and then design test cases to prevent such errors. The goal is not to test the specifications, but to derive test cases that would detect implementations of the mutated specifications. Hence, our strategy is to prevent the IUT to conform to erroneous specifications.

In our previous work, we have shown that the notion of refinement may be used to define the properties of such fault-based test cases. In [17] we discussed mutation testing in Back's refinement calculus. More recently, we developed a tool for generating test cases from mutated OCL specifications based on these ideas [18]. In the following, we will reformulate these ideas for IOLTSs and test purposes.

3.2 Relating Test Purposes, Test Cases and Specifications

In the context of software specification, programs are linked to their specification by a refinement relation. The conformance relation in Definition 2 is an example of such a refinement relation between IOLTS models. In the context of black-box testing, we can imagine a similar relation between test cases and test purposes, since test purposes are specifications of test cases [10]: The relation

$$\mathit{refines}(TC, TP, S),$$

where TC is a test case, TP a test purpose and S a specification, captures the property when a test case is a refinement of a test purpose in the context of a specification. Note that it is necessary to include the specification, since a test purpose is only meaningful together with a specification. In TGV the *refines* relation is defined by the properties of test cases presented in Section 2.3. Hence, given the test case generation algorithm of TGV $\mathit{generate}_{TGV}(TP, S)$ we have

$$\mathit{refines}(\mathit{generate}_{TGV}(TP, S), TP, S)$$

This relation expresses the fact that the test purpose TP and the specification S are consistent such that TGV is able to generate a proper test case. Hence, we abbreviate it as

$$consistent_{TGV}(TP, S) \quad =_{df} \quad refines(generate_{TGV}(TP, S), TP, S)$$

3.3 Fault-Based Test Purposes

We will now use this consistency (refinement) relation to express the property of the test purposes (test cases) we are interested in. Imagine a specification S and a mutated version S^m which has been modified by inserting a fault into S. Our goal is to generate a test case that is able to distinguish an implementation of S and S^m. For generating such a test case TC we need to formulate an appropriate test purpose TP that would guarantee that only such discriminating test cases are generated. Hence, the test purpose must not abstract away from the erroneous part in S^m, but must detect the differences between S and S^m. This can be formally expressed via the consistency relation:

$$consistent_{TGV}(TP, S) \text{ and } \neg consistent_{TGV}(TP, S^m)$$

This conjunction expresses the fact that the test purpose is able to distinguish between S and S^m, since it is inconsistent with the later. This means that the test goal can be achieved with respect to one specification, but not with the other. Consequently, a test case generated from such a test purpose TP will be able to distinguish between implementations of S and S^m. These are the test purposes we are going to generate.

One well-known challenge of mutation testing are equivalent mutants. An equivalent mutant occurs when an introduced syntactical change in the model does not represent an observable fault, hence the original S and a mutant S^m are observably equivalent ($S \approx S^m$). Without yet defining the kind of observable equivalence relation, we may formalize that if a discriminating test purpose does not exists, we have observable equivalence:

$$\nexists\, TP : (consistent_{TGV}(TP, S) \ \wedge \ \neg consistent_{TGV}(TP, S^m)) \ \Rightarrow \ (S \approx S^m)$$

This leads us immediately to the property that will be the basis for our test purpose generation:

$$(S \not\approx S^m) \ \Rightarrow \ \exists\, TP : (consistent_{TGV}(TP, S) \ \wedge \ \neg consistent_{TGV}(TP, S^m))$$

In the next section we will see that an equivalence checker for finite LTSs can be used to generate test purposes from equivalence counter examples.

4 A Technique for Fault-Based Testing

In this section, we present the technique for generating test purposes from injected faults. As indicated in the previous section, an equivalence checker for labelled transition systems forms the key technology of the generation process.

4.1 The Process

We use an equivalence checker of the CADP tools which also contain TGV. We use LOTOS as a specification language, but other input formats that can be converted to the internal CADP format for a LTS are possible.

We first present an overview of the essential steps in the process and then discuss the details.

1. Model a system to be tested in LOTOS with explicit input and output actions. Select a mutation operator for LOTOS and create a mutant L^m from the original model L.
2. Generate an IOLTS S_τ and S_τ^m from the specifications L and L^m respectively (using CADP-Caesar).
3. Simplify the rather large IOLTS S_τ and S_τ^m to obtain S and S^M using the Safety Equivalence relation (using CADP-Aldebaran).
4. Check the reduced IOLTS S and S^m for Strong Bisimulation (using CADP-Aldebaran).
5. The equivalence check gives
   (a) True: S^m is an equivalent mutant (no fault), no test purpose can be generated. Study the cause of equivalency. There might be a redundancy in the model!
   (b) False: CADP-Aldebaran issues a diagnosis (counterexample): a discriminating sequence c.
6. Add one more valid transition from S to the counterexample c (if any) in order to create a valid path which can discover the injected error. This sequence forms the wanted test purpose.
7. Generate a test case from the discriminating test purpose (using CADP-TGV).
8. Test the IUT with this test case to prevent that the IUT conforms to the faulty specification L^m.
9. Repeat this for every interesting mutation possible.

The IOLTS generated from LOTOS needs to be simplified, since it contains many redundant internal τ actions. As mentioned in Section 2.2 the test case generation process does involve visible actions only. CADP provides a simplification tool that removes all τ actions and generates an LTS that is equivalent with respect to safety properties. This Safety Equivalence relation is defined as follows [19]:

Definition 5 (Safety Equivalence). *Let $S = (Q, A_\tau, T, q_0)$ be a Labelled Transition System and let $p, r \in Q$. Safety equivalence is defined as*

$$p \approx^{saf} r \quad =_{df} \quad p \sqsubseteq^{saf} r \wedge r \sqsubseteq^{saf} p$$

The relation $\sqsubseteq^{saf}$ may be characterized as weak simulation:

$$p \sqsubseteq^{saf} r \textbf{ iff } \forall a \in A_\tau, \forall p' : (p \xrightarrow{\tau^* a} p' \Rightarrow \exists r' \bullet (r \xrightarrow{\tau^* a} r' \wedge p' \sqsubseteq^{saf} r'))$$

Then, since all τ actions are removed, a strong bisimulation check can be applied to determine the observational (non-)equivalence of both models. In the more likely case of non-equivalence, CADP generates a sequence of actions leading to a state where both behaviors deviate. Then, a test purpose IOLTS is produced which has as its only trace this sequence plus the next discriminating action of the original. This test purpose serves to generate a test case that will distinguish an implementation of the original from an implementation of the faulty mutant. This process has to be repeated for all faults one wishes to test for. Only one fault per mutant is injected (coupling effect assumption).

4.2 Mutation Operators

At the heart of this fault-based testing technique is the set of faults that are injected into a given model. These faults represent the errors one is able to anticipate and that are to be prevented by the generated test cases. They also form the basis for the coverage criterion: For each injected fault (that can be observed) there must exist a test case in the test suite able to detect it. Consequently, the set of injected faults is critical.

A common strategy in mutation testing is to define a set of mutation operators for the language in use. A mutation operator syntactically transforms a language construct, by exchanging, deleting or adding parts of it. Once the mutation operators are defined the mutants can be generated systematically or even automatically. We have defined such a set of mutation operators for Full LOTOS (see Table 1). Most of these operators are not special to LOTOS and have been considered before. ORO, SNO, ENO, LRO, RRO, MCO, ACO, STO and ASO are taken from [20]. EDO, ESO, ERO, EIO, SOR, POR, MRO, CRO, USO, HDO and PRO are taken from [21] (mutation operators for CSP). Others, like in [22], used subsets of these. We have newly added the PSP, PRP, ESP, ERP, SSP and SRP operators which are special to LOTOS.

Mutation operators are not the only source for injected faults. This is especially true on the modeling level. In security testing known vulnerabilities might be modeled as mutations. Another source for faults are common semantic misinterpretations of requirements or of a modeling language. One might imagine a set of mutation operators for UML constructs that are ambigous. Test cases could be designed such that a common interpretation of these models is enforced.

5 Web Server Case Study

In this section we report an a case study on testing web servers that serves to demonstrate that our technique is applicable. The aim was to test the correct implementation of parts of the HTTP protocol in the Apache web server. We focused on the GET-Method responsible for retrieving pages and limited ourselves to single client-server connections.

The source for the LOTOS model was the Internet standard RFC 2616 (Request for Comments). RFC 2616 specifies the syntax of the HTTP protocol in

Table 1. Mutation Operators for Extended LOTOS

Symbol	Mutation Operators	Description
EDO	Event Drop Operator	Eliminate one of the events from the process definition
ESO	Event Swap Operator	Change order of the 2 neighbouring events
ERO	Event Replacement Operator	Replace event by other events
EIO	Event Insertion Operator	Inserts one event after each event in the process definition
SOR	Sequential Operator Replacement	Replace the sequential composition operator (enabling and disabling) >> and [>
POR	Process Operator Replacement	Replace the operator on processes (Parallel composition general case, pure interleaving and full synchronization) ||, |[]| and |||
MRO	Message Replacement Operator	Replace the message of each communication channel with other message
CRO	Channel Replacement Operator	Replace the channel with other channels within the process definition
USO	Unobservable Sequence Operator	Change the action prefix from unobservable to observable
HDO	Hiding Delete Operator	Delete an event from hide definition
PRO	Process Replacement Operator	Replace the process name with stop or exit events
SEO	Stop and Exit interchange Operator	Interchange the Stop and Exit events
PSP	Process Swap Parameter	Change order of the two neighbouring parameters in process calls
PRP	Process Replace Parameter	Replace one parameter with other in process calls
ESP	Exit Swap Parameter	Change order of the two neighbouring parameters in Exit operator calls
ERP	Exit Replace Parameter	Replace one parameter with other in Exit operator calls
SSP	Sequential composition Swap Parameter	Change order of the two neighbouring parameters in Parameterized Sequential Composition operator calls
SRP	Sequential composition Replace Parameter	Replace one parameter with other in Parameterized Sequential Composition operator calls
ORO	Operand Replacement Operators	Replace an operand (variable or constant) by another syntactically legal operand in data type declarations
SNO	Simple Expression Negation Operators	Replace a simple expression by its negation
ENO	Expression Negation Operators	Replace an expression by its negation
LRO	Logical Operators Replacement	Replace a logical operator (and, or, not) by another
RRO	Relational Operators Replacement	Replace a relational operator ($<, \leq, >, \geq, =, \neq$ on basic types or whatever is declared in data type declarations) by any other except its opposite
MCO	Missing Condition Operators	Delete conditions from conjunctions, disjunctions and implications
ACO	Adding Condition Operators	Add conditions from conjunctions, disjunctions and implications
STO	Stuck At Operators	Replace a simple expression with 0 or 1
ASO	Associative Shift Operators	Change the association between variables

BNF and describes the semantics in natural language (English). Our model consists of two LOTOS processes, the client and the server, running in parallel. The client is issuing a request message and then waits for a response message from the server process. The request message contains three parts, each one modeled as an action: (1) a Request Line (with a Method (here GET), a URI and the HTTP-version), (2) a Request Header and (3) an optional Request Body. The Request Header facilitates conditional requests, like e.g. header If_Modified_Since supports a restricted download of pages that have been recently updated. The Response message of the server contains three parts, too: (1) a Status Line (with the HTTP-version, a Status Code and a Reason), a Response Header (with information about the web page) and a Response Body (which contains the web page in most cases).

The choice of the level of granularity of the actions is a pragmatic one. Which part of the protocol is modeled as an action depends on the actual testing strategy and how the actions are easily mapped to real interactions with the web server. For example, the three parts of the Request Line have been merged into a single action, since we were not interested in testing variations of URI's and HTTP-versions.

Given the mutation operators in Table 1, almost 1500 mutants were derived from the HTTP model. Table 2 shows that a relative high number of equivalent

Table 2. Number of generated mutants

Symbol	Mutation Operator	No.Mutants	No.Equiv Mutants
EDO	Event Drop Operator	57	0
ESO	Event Swap Operator	15	0
ERO	Event Replacement Operator	65	5
EIO	Event Insertion Operator	63	7
SOR	Sequential Operator Replacement	17	7
POR	Process Operator Replacement	5	1
MRO	Message Replacement Operator	97	0
CRO	Channel Replacement Operator	46	0
USO	Unobservable Sequence Operator	2	0
HDO	Hiding Delete Operator	0	0
PRO	Process Replacement Operator	6	0
SEO	Stop and Exit interchange Operator	45	0
PSP	Process Swap Parameter	41	10
PRP	Process Replace Parameter	59	0
ESP	Exit Swap Parameter	44	15
ERP	Exit Replace Parameter	48	0
SSP	Sequential composition Swap Parameter	10	0
SRP	Sequential composition Replace Parameter	12	0
ORO	Operand Replacement Operators	154	77
SNO	Simple Expression Negation Operators	154	77
ENO	Expression Negation Operators	78	38
LRO	Logical Operators Replacement	80	8
RRO	Relational Operators Replacement	15	0
MCO	Missing Condition Operators	41	18
ACO	Adding Condition Operators	69	27
STO	Stuck At Operators	220	30
ASO	Associative Shift Operators	48	22
	Total	1491	342

mutants was obtained, especially by the ORO and SNO operators. The reason is that the synchronized product of the GET request of the client and the more complete specification of the server makes large parts of the HTTP model redundant. For example, the server is ready to listen to all kinds of Methods, but only one (GET) is actually requested by the client.

The large number of appr. 1150 non-equivalent mutants shows that a testing with mutation operators can only be done if the testing process is completely automated. However, complete automation was not the goal of this case study. Hence, we selected about 100 interesting mutations that were partly reflecting ambiguities in the HTTP standard. From these we generated the test purposes according to the process described in the previous section and used TGV to generate the test cases.

The implementation under test was our institute's Apache Web Server 2.0.40 for Red Hat Linux with HTTP/1.1 protocol. The tests have been carried out manually via a telnet session to Port 80 of the web server's URL. This is possible since the HTTP protocol is ASCII based.

We did not expect to find major flaws in the Apache Web Server, since it has been widely used since years. However, we found a case where Apache behaves unexpectedly. As mentioned above, conditional requests can be formed by adding header fields. They serve to control the caching done by a proxy server. The combination of several such header fields is underspecified in the standard. However, on page 56 of RFC 2616 the standard says:

> "An HTTP/1.1 origin server, upon receiving a conditional request that includes both a Last-Modified date (e.g., in an If-Modified-Since or If-Unmodified-Since header field) and one or more entity tags (e.g., in an

> If- Match, If-None-Match, or If-Range header field) as cache validators, *must not* return a response status of 304 (Not Modified) unless doing so is consistent with all of the conditional header fields in the request."

Entity Tags and Last-Modifed Times are metainformations used to find out whether a cache entry is an equivalent copy of an entity. The description in the RFC is ambiguous, but it indicates that priority should be given to the If-Match, If-None-Match, and If-Range header fields. The first tests showed that the Apache developers shared our interpretation:

ID	*Header 1 satisfied?*	*Header 2 satisfied?*	*Response Status*
1	If-Match = true	If-Modified-Since = true	OK (200)
2	If-Match = true	If-Modified-Since = false	Not Modified (304)
3	If-Match = false	If-Modified-Since = true	Precondition Fail (412)
4	If-Match = false	If-Modified-Since = false	Precondition Fail (412)
5	If-Match = false	If-Unmodified-Since = true	Precondition Fail (412)
6	If-Match = false	If-Unmodified-Since = false	Precondition Fail (412)
7	If-None-Match = false	If-Unmodified-Since = false	Precondition Fail (412)

Note that test cases 3–7 respond with code 412 following the interpretation that the response of the Match header has higher priority. However, the following test cases suddenly deviate from this pattern:

ID	*Header 1 satisfied?*	*Header 2 satisfied?*	*Response Status*
8	If-None-Match = false	If-Modified-Since = false	Not Modified (304)
9	If-None-Match = false	If-Unmodified-Since = true	Not Modified (304)

This is a rather unexpected response of Apache which does not seem to be consistent with the If-Match cases.

6 Conclusions

We have presented a mutation testing technique for generating test purposes. The theory relating fault-based test purposes to mutated specifications is very similar to our previous testing theory for the refinement calculus and OCL. There, a test case t for distinguishing implementations of S and S^m had to satisfy $\mathit{refines}(S, t)$ and $\neg \mathit{refines}(S^m, t)$, with *refines* being defined via weakest preconditions (refinement calculus [17]) and implication (OCL [18]). Hence, our fault-based IOLTS theory is a further instantiation of this refinement property and demonstrates its generality.

To our present knowledge this is the first work on generating test purposes via specification mutation. However, others have worked on mutation testing on the specification level before. Most of them either focus on testing the specification or on generating test cases directly. To our present knowledge Budd and Gopal were the first [23]. They applied a set of mutation operators to specifications given in predicate calculus form. The method relies on having a working implementation generating output.

Tai and Su [24] propose algorithms for generating test cases that guarantee the detection of operator errors, but they restrict themselves to the testing of singular Boolean expressions, in which each operand is a simple Boolean variable that cannot occur more than once. Tai [25] extends this work to include the detection of Boolean operator faults, relational operator faults and a type of fault involving arithmetic expressions. However, the functions represented in the form of singular Boolean expressions constitute only a small proportion of all Boolean functions.

Stocks applied mutation testing to Z specifications [16]. He presented the criteria to generate test cases to discriminate mutants, but did not automate his approach. Woodward investigated mutation operators for algebraic specifications [26].

More recently, Simon Burton presented a fault-based test case generator for Z specifications [27]. He uses a combination of a theorem prover and a collection of constraint solvers. The theorem prover generates a disjunctive normal form, simplifies the formulas and helps in formulating different testing strategies.

Black et al. studied mutation operators using the SMV model checker [28]. However, they do not consider test purposes. A group in York has recently started to use fault-based techniques for validating their CSP models [21]. Their aim is not to generate test cases, but to study the equivalent mutants. Similar research is going on in Brazil with an emphasis on protocol specifications written in the Estelle language [29].

Wimmel and Jürjens [30] use mutation testing on specifications to extract those interaction sequences that are most likely to find security issues. This work is closest to ours, since they generate test cases for finding faults in concurrent systems.

Our approach needs further evaluation. Its efficiency compared to structural model-based testing techniques needs to be analysed. Especially, the optimal choice of mutation operators deserves our attention. The case study indicates, for example, that some mutation operators are more likely to generate equivalent mutants than others.

The presented technique is specific to the TGV test case generator and similar tools. However, the theoretical discussion of the properties of fault-based test purposes has been included to make the result more widely applicable. For example, [10] discusses test purposes for the B specification language. Consequently, a similar technique could be developed for B and other model-oriented specification languages.

References

1. Gaudel, M.: Testing can be formal, too. In: TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, Springer-Verlag (1995) 82–96
2. Jéron, T., Morel, P.: Test Generation Derived from Model-checking. In: Computer Aided Verification (CAV'99). Volume 1633 of Lecture Notes in Computer Science., Springer (1999)

3. Fernandez, J.C., Garavel, H., Kerbrat, A., Mounier, L., Mateescu, R., Sighireanu, M.: CADP: a protocol validation and verification toolbox. In Alur, R., Henzinger, T.A., eds.: Proceedings of the Eighth International Conference on Computer Aided Verification CAV. Volume 1102., Springer Verlag (1996) 437–440
4. Tretmans, J.: Test Generation with inputs, outputs and repetitive quiescence. Software: Concepts and Tools **17** (1996) 103–120
5. ISO: ISO/IEC 9646-3: Information technology - OSI - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN). Technical report, iso.ch (1998)
6. ISO: ISO/IEC 9646-1: Information technology - OSI - Conformance testing methodology and framework - Part 1: General Concepts. Technical report, iso.ch (1994)
7. Grabowski, J., Hogrefe, D., Nahm, R.: Test case generation with test purpose specification by MSC's. In: SDL'93, the 6th SDL Forum, Elsevier Science (1993) 253–266
8. de Vries, R.G., Tretmans, J.: Towards formal test purposes. In Brinksma, E., Tretmans, J., eds.: Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES'01), Aalborg, Denmark, August 25, 2001. Number NS-01-4 in BRICS Notes Series, BRICS, Department of Computer Science, University of Aarhus (2001) 61–76
9. Grieskamp, W., Tillmann, N., Campbell, C., Schulte, W., Veanes, M.: Action Machines — Towards a Framework for Model Composition, Exploration and Conformance Testing Based on Symbolic Computation. In Cai, K.Y., Ohnishi, A., Lau, M., eds.: QSIC 2005, Proceedings of the Fifth International Conference on Quality Software, Melbourne, Australia, September 19–20, 2005, IEEE Computer Society (2005) 72–79
10. Ledru, Y., du Bousquet, L., Bontron, P., Maury, O., Oriat, C., Potet, M.L.: Test purposes: adapting the notion of specification to testing. In Feather, M., Goedicke, M., eds.: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE2001), San Diego, CA, USA, 26–29 November 2001, IEEE Computer Society (2001) 127–134
11. Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering **3** (1977) 279–290
12. DeMillo, R., Lipton, R., Sayward, F.: Hints on test data selection: Help for the practicing programmer. IEEE Computer **11** (1978) 34–41
13. Wong, W.E., ed.: Mutation Testing for the New Century. Kluwer Academic Publishers (2001)
14. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications: A theory and a tool. Software Engineering Journal **6** (1991) 387–405
15. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In Woodcock, J., Larsen, P., eds.: Proceedings of FME'93: Industrial-Strength Formal Methods, International Symposium of Formal Methods Europe, April 1993, Odense, Denmark, Springer-Verlag (1993) 268–284
16. Stocks, P.A.: Applying formal methods to software testing. PhD thesis, Department of computer science, University of Queensland (1993)
17. Aichernig, B.K.: Mutation Testing in the Refinement Calculus. Formal Aspects of Computing Journal **15** (2003) 280–295
18. Aichernig, B.K., Salas, P.A.P.: Test case generation by OCL mutation and constraint solving. In Cai, K.Y., Ohnishi, A., Lau, M., eds.: QSIC 2OO5, Proceedings of the Fifth International Conference on Quality Software, Melbourne, Australia, September 19-21, 2005, IEEE Computer Society Press (2005) 64–71

19. Fernandez, J.C.: Aldébaran: A tool for verification of communicating processes. Technical report, Technical Report Spectre C14, LGJ-IMAG Grenoble (1989)
20. Black, P., Okun, V., Yesha, Y.: Mutation operators for specifications. In: Proceedings of 15th IEEE International Conference on Automated Software Engineering (ASE'00). (2000) 81–88
21. Srivatanakul, T., Clark, J., Stepney, S., Polack, F.: Challenging formal specifications by mutation: a CSP security example. In: Proceedings of APSEC 2003: 10th Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand, December, 2003, IEEE (2003) 340–351
22. Offutt, J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology **5** (1996) 99–118
23. Budd, T., Gopal, A.: Program testing by specification mutation. Comput. Lang. **10** (1985) 63–73
24. Tai, K.C., Su, H.K.: Test generation for Boolean expressions. In: Proceedings of the Eleventh Annual International Computer Software and Applications Conference (COMPSAC). (1987) 278–284
25. Tai, K.C.: Theory of fault-based predicate testing for computer programs. IEEE Transactions on Software Engineering **22** (1996) 552–562
26. Woodward, M.: Errors in algebraic specifications and an experimental mutation testing tool. Software Engineering Journal (1993) 211–224
27. Burton, S.: Automated Testing from Z Specifications. Technical Report YCS 329, Department of Computer Science, University of York (2000)
28. Black, P., Okun, V., Yesha, Y.: Mutation of model checker specifications for test generation and evaluation. In: Mutation testing for the new century. Kluwer Academic Publishers (2001) 14–20
29. de Souza, S., Fabbri, J.M.S., de Souza, W.: Mutation testing applied to Estelle specifications. Software Quality Journal **8** (1999) 285–301
30. Wimmel, G., Jürjens, J.: Specification-based test generation for security-critical systems using mutations. In George, C., Huaikou, M., eds.: Proceedings of ICFEM'02, the International Conference of Formal Engineering Methods, October 21–25, 2002, Shanghai, China. LNCS, Springer-Verlag (2002) 471–482