Information and Software Technology 51 (2009) 1601–1617

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Using coverage to automate and improve test purpose based testing ☆

Martin Weiglhofer *, Gordon Fraser, Franz Wotawa

Institute for Software Technology, Graz University of Technology, 8010 Graz, Austria

ARTICLE INFO

Article history:
Available online 5 July 2009

Keywords:
Coverage criteria
Test purposes
LOTOS
TGV
Model-based testing

ABSTRACT

Test purposes have been presented as a solution to avoid the state space explosion when selecting test cases from formal models. Although such techniques work very well with regard to the speed of the test derivation, they leave the tester with one important task that influences the quality of the overall testing process: test purposes have to be formulated manually. In this paper, we present an approach that assists a test engineer with test purpose design in two ways: it allows automatic generation of coverage based test suites and can be used to automatically exercise those aspects of the system that are missed by hand-crafted test purposes. We consider coverage of LOTOS specifications, and show how labeled transition systems derived from such specifications have to be extended in order to allow the application of logical coverage criteria to LOTOS specifications. We then show how existing tools can be used to efficiently derive test cases and suggest how to use the coverage information to minimize test suites while generating them.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Test purposes can be used to create test cases efficiently, even when large models are used for test case generation. A test purpose is a formal representation of what the tester intends to exercise. Given a test purpose, only a limited part of a test model needs to be considered which attenuates the state space explosion problem. Unfortunately, this still leaves the tester with the task of writing test purposes, which might turn out rather difficult if thorough testing is required. In fact, despite all formal conformance relations, the real quality of the test suites depends on the ability of the tester to formulate suitable test purposes. For example, du Bousquet et al. [2] report that even after 10 h of manual test purpose design they failed to find a set of test purposes that would detect all mutants of a given implementation.

In this paper, we present a set of generic test purposes based on coverage criteria for LOTOS [3] specifications. Such an approach offers the advantages of established test selection strategies together with the efficiency of test purpose based test derivation algorithms. This combination is not directly possible; test purposes are applied to an underlying model formalism, which is mostly the labeled transition system (LTS). We are not simply looking at coverage of labels or transitions of the LTS, but aim to cover important aspects of the underlying specification, which is typically given in some formal language like LOTOS. However, not all relevant information contained in a LOTOS specification is directly reflected in the LTS.

We define coverage criteria for LOTOS specifications based on well-known structural criteria and criteria for logical conditions, and show how the derived LTS can be extended to contain the necessary information. This additional information is used within our test purposes for different coverage criteria; given our test purposes available tools such as TGV [4] can be used to automatically derive test suites. The technique can be applied by itself as a sufficient testing method, but it can also support a test engineer to detect and automatically exercise those parts of a system that manually specified test purposes do not cover.

This paper is based on our previous work [1], which it extends with:

- two new coverage criteria, i.e., process coverage and modified condition/decision coverage, for LOTOS specifications,
- an improved discussion of weak and strong coverage including formal definitions,
- a discussion of how to complement test suites derived from manually designed test purposes,
- results obtained by applying our approach to an additional specification, and
- results for complementing manually designed test purposes.

This paper is organized as follows: first, we introduce the necessary preliminaries of LOTOS and LTS based testing in Section 2. Then, we define coverage criteria for LOTOS in Section 3, show how the LOTOS specification has to be extended such that the resulting LTS

☆ A preliminary version of this paper was presented at the 8th International Conference on Quality Software (QSIC 2008) [1].

* Corresponding author.
E-mail addresses: weiglhofer@ist.tugraz.at (M. Weiglhofer), fraser@ist.tugraz.at (G. Fraser), wotawa@ist.tugraz.at (F. Wotawa).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.06.008

contains all necessary information, and describe how the coverage test purposes are derived. In Section 4, we present results of an evaluation conducted on two different protocol specifications. Finally, we review related work in Section 5 and conclude the paper in Section 6.

2. Preliminaries

The approach presented in this paper considers test case generation for LOTOS specifications. Semantically, LOTOS specifications can be interpreted as labeled transition systems. Test case generation for labeled transition systems is a much studied field of research. There is a well-defined theory underlying such approaches, and mature research prototypes (e.g., TGV [4], TORX [5]) have been presented. Consequently, this is also the approach we have chosen. In this section we give a short introduction to the language LOTOS and then briefly review the notion of test cases and test purposes as used by TGV for test generation with respect to Tretmans's input/output conformance (ioco) theory [6]. The example specification given in this section will be used in the subsequent sections to describe our testing approach.

2.1. A brief introduction to LOTOS and labeled transition systems

This section contains a brief introduction to LOTOS; for a detailed introduction to the LOTOS specification language we refer the reader to the work of Bolognesi and Brinksma [7].

The language of temporal ordering specification (LOTOS) is an ISO standard [3]. LOTOS comprises two components: The first is based on the Calculus of Communication Systems [8] (CCS) and deals with the behavioral description of a system, which is given in terms of processes, their behavior, and their interactions. The second component of LOTOS specifications is used to describe data structures and value expressions, and is based on the abstract data type language ACT ONE [9].

The basic elements of a LOTOS specification are processes with a certain behavior expressed in terms of actions. An action is an expression over a process's gates, possibly equipped with values. Table 1 lists some of the main elements used to compose the behavior of a process.

The ";" operator is used to express sequential composition. For example, the behavior $a;b$ in a process with the gates a and b expresses that first a and then subsequently b happens. A guarded behavior `[guard]->behavior` is only executed if the guard evaluates to true. For example, in the specification shown in Fig. 2 Line 29 can only be executed if the size of the stack s is equal to one.

Table 1
Excerpt of LOTOS behavior elements.

Syntax	Meaning
`action; behavior`	Action followed by a behavior
`[guard] -> behavior`	Guarded behavior
`behaviorl []` `  behavior2`	Choice
$B_1 \vert\vert\vert B_2$	Interleaving of two behaviors
$B_1 \vert\vert B_2$	Behaviors are synchronized on all actions
$B_1 \vert[g_1,\ldots,g_n]\vert B_2$	Behaviors are partially synchronized on the gates $g_1,\ldots,g_n$
`behaviorl [>` `  behavior2`	Disabled by second behavior
`behaviorl >>` `  behavior2`	First enables second
`exit`	Exit
`proc[gate,..](val,..)`	Process instantiation
`(behavior)`	Grouping

A choice between two behaviors B_1 and B_2 is expressed by $B_1 [] B_2$. For example, let the gates of a process be `milk` and `tee`, then the expression `milk[]tee` allows one to chose between the events tee and coffee.

The LOTOS construct $B_1 [> B_2$ states, that choosing the first action of the behavior B_2 disables the execution of the behavior B_1. For example, Line 3 of Fig. 2 expresses that the `Main` process is executed unless the event `ui !quit` is executed. After `ui !quit` the execution continues in the right branch of the $[>$ operator. In the case of the specification of Fig. 2 this means that the specification terminates (`exit`).

The enabling operator (>>) within a LOTOS specification states that a successful execution of the first process enables the subsequent behavior block. For example consider Line 10 of Fig. 2. This line states, that after successful termination of the `DisplayStack` process the behavior of the specification is obtained by recursively instantiating the `Main` process.

An interleaving expression $B_1 \vert\vert\vert B_2$ expresses that the two behaviors B_1 and B_2 are executed without any synchronization, i.e. the actions may be executed in an arbitrary order. For example, let a_1, a_2 and b_1 be actions, then the specification $(a_1;a_2)\vert\vert\vert(b_1)$ has the following observable behaviors: $\langle a_1,a_2,b_1\rangle$, $\langle a_1,b_1,a_2\rangle$, $\langle b_1,a_1,a_2\rangle$. Contrary, partial synchronization $(B_1 \vert[g_1,\ldots,g_n]\vert B_2)$ expresses synchronization only on the specified gates $g_1,\ldots,g_n$. A synchronized action can only be executed if both synchronized behavioral blocks offer the action, i.e. let a_1, a_2 and b_1 be actions, then the specification $(a_1;a_2)\vert[a_2]\vert(a_2;b_1)$ has only the following action sequence: $\langle a_1,a_2,b_1\rangle$. Finally, full synchronization, i.e. $B_1 \vert\vert B_2$, requires that all actions of B_1 and B_2 are synchronized.

Process instantiation allows one to use defined processes. For example, Line 3 of Fig. 2 instantiates the `Main` process with the gates *ui* and *out* and an empty stack, i.e. *nil*, as parameter.

An abstract data type is given in terms of sorts, operations and equations. An example showing the basic elements of data type definitions is given in Fig. 1. This data type comprises the base element nil (Line 5), which denotes an empty stack, and the four prefix operators `push`, `pop`, `top`, and `size`. For example, the signature of the `push` operator on Line 6 shows that this operator takes a natural number (Nat) and a stack and returns a new stack. The equation part of this data type definition states that for any stack *st* and for any natural number *n*, popping a previously pushed element of a stack *st* results into *st* (Line 13).

The semantics of a LOTOS specification is given as a labeled transition system.

Definition 1 (*Labeled transition system*). A labeled transition system (LTS) is a tuple $M = \left(Q^M, A^M \cup \{\tau\}, \rightarrow_M, q_0^M\right)$, where Q^M a finite set of states, A^M a finite alphabet and $\tau \notin A^M$ 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 are interested in generating test cases for testing conformance with respect to input output conformance (ioco) as defined by Tretmans [6]. Hence, the LTS is extended with input and output information before test generation; this gives an input output labeled transition system.

Definition 2 (*Input output labeled transition system*). An input output labeled transition system (IOLTS) is an LTS $M = \left(Q^M, A^M \cup \{\tau\}, \rightarrow_M, q_0^M\right)$ where A^M is partitioned into two disjoint sets $A^M = A_I^M \cup A_O^M$, where A_I^M and A_O^M are input and output alphabets, respectively.

We use the following common notations for LTSs and for IOLTSs.

Definition 3. Let M be a labeled transition system $M = \left(Q^M, A_I^M \cup A_O^M \cup \{\tau\}, \rightarrow_M, q_0^M\right)$ and let $q, q', q_0, q_1, \ldots, q_n \in Q^M, a, a_1, \ldots, a_n \in A_I^M \cup A_O^M$, and $\sigma \in \left(A_I^M \cup A_O^M\right)^*$, then

```
1  type Stack is NaturalNumber
2     sorts
3        Stack
4     opns
5        nil  : -> Stack
6        push : Nat, Stack -> Stack
7        pop  : Stack -> Stack
8        top  : Stack -> Nat
9        size : Stack -> Nat
10    eqns
11       forall st: Stack, n: Nat
12          ofsort Stack
13             pop(push(n,st)) = st;
14          ofsort Nat
15             size(nil) = 0;
16             size(push(n,st)) = Succ(size(st));
17             top(push(n,st)) = n;
18 endtype
```

Fig. 1. A simple stack data type providing typical stack operations.

$$
\begin{aligned}
q \xrightarrow{a}_M q' &=_{df} (q,a,q') \in \rightarrow_M \\
q \xrightarrow{a}_M &=_{df} \exists q' \cdot (q,a,q') \in \rightarrow_M \\
q \overset{\epsilon}{\Rightarrow} q' &=_{df} (q = q') \vee \exists q_0,\ldots,q_n \cdot (q = q_0 \xrightarrow{\tau}_M q_1 \wedge \cdots \wedge q_{n-1} \xrightarrow{\tau}_M q_n = q') \\
q \overset{a}{\Rightarrow} q' &=_{df} \exists q_1,q_2 \cdot q \overset{\epsilon}{\Rightarrow}_M q_1 \xrightarrow{a}_M q_2 \overset{\epsilon}{\Rightarrow}_M q' \\
q \overset{a_1,\ldots,a_n}{\Rightarrow} q' &=_{df} \exists q_0,\ldots,q_n \cdot q = q_0 \overset{a_1}{\Rightarrow}_M q_1,\ldots,q_{n-1} \overset{a_n}{\Rightarrow}_M q_n = q' \\
q \overset{\sigma}{\Rightarrow} &=_{df} \exists q' \cdot q \overset{\sigma}{\Rightarrow} q'
\end{aligned}
$$

We use $init(q)$ to denote the actions enabled in state q and $traces(q)$ to denote the traces starting in state q. Informally, a *trace* of an LTS is a finite sequence of actions allowed by the transition relation. Furthermore, we denote the states reachable by a particular trace σ by q **after** σ. More precisely,

Definition 4 [10]. Let M be a labeled transition system $M = \left(Q^M, A_I^M \cup A_O^M \cup \{\tau\}, \rightarrow_M, q_0^M\right)$ and let $q \in Q^M, Q \subseteq Q^M$, and $\sigma \in \left(A_I^M \cup A_O^M\right)^*$, then

$$
\begin{aligned}
init(q) &=_{df} \left\{ a \in A_I^M \cup A_O^M \cup \{\tau\} | q \xrightarrow{a}_M \right\} \\
traces(q) &=_{df} \left\{ \sigma \in \left(A_I^M \cup A_O^M\right)^* | q \overset{\sigma}{\Rightarrow} \right\} \\
q \ \mathbf{after}_M \ \sigma &=_{df} \left\{ q' |\ q \overset{\sigma}{\Rightarrow}_M q' \right\} \\
Q \ \mathbf{after}_M \ \sigma &=_{df} \bigcup_{q \in Q} (q \ \mathbf{after}_M \ \sigma)
\end{aligned}
$$

Note that $init(q)$ may contain τ actions while $traces(q)$ only comprises observable behavior, i.e. no τ actions, starting at state q. When talking about traces one is not interested in unobservable behavior. Contrary, when looking at the actions initially enabled in state q one wants to know if there is a τ labeled transition enabled in q. Because in that case the system may internally move to another state. This is not relevant with respect to the traces of q.

An LTS $M = \left(Q^M, A_I^M \cup A_O^M \cup \{\tau\}, \rightarrow_M, q_0^M\right)$ is *deterministic* if for any sequence of actions, starting at the initial state, there is at most one successor state, i.e. $\forall \sigma \in \left(A_I^M \cup A_O^M\right)^* : |q_0^M \ \mathbf{after}_M \sigma| \leqslant 1$, where $|X|$ denotes the cardinality of the set X.

Labeled transition systems can be composed using parallel composition, which is denoted by the parallel composition operator $\|$. Given two labeled transition systems M_1 and M_2, then the operational semantics of $M_1 \| M_2$ is defined by the following inference rule:

$$
\frac{q^{M_1} \xrightarrow{\alpha}_{M_1} q^{M_1\prime}, q^{M_2} \xrightarrow{\alpha}_{M_2} q^{M_2\prime}}{q^{M_1} || q^{M_2} \xrightarrow{\alpha}_{M_1 \| M_2} q^{M_1\prime} || q^{M_2\prime}}
$$

As this inference rule illustrates, the parallel composition $M_1 \| M_2$ comprises only parts common to both M_1 and M_2. Note that LOTOS's $\|$ operator has a similar semantics, i.e. synchronization happens on all actions of the synchronized behaviors.

2.2. Example LOTOS specification of a stack calculator

Fig. 2 shows a LOTOS specification of a simple stack calculator, which will be used to illustrate coverage criteria in the subsequent section. The specification uses both behavioral description elements and data type elements. The stack data type is specified in Fig. 1. For the sake of brevity we do not show the data type definitions for Boolean elements and for natural numbers in this paper; our specification simply relies on the types "Boolean" and "NaturalNumber" stated in the appendix of the LOTOS ISO standard [3]. In addition, we omit the definition of our "Operator" type, which only provides the two base elements `add` and `display` and comparison operators for those two elements.

The specification of the stack calculator uses the two gates `ui` for user input and `out` for system output and consists of two processes. The main process (Lines 5–25) takes the current stack `s` as argument and reads either an operator (Line 7) or an operand (Line 24), i.e., a natural number, from the user. If the user enters a natural number, the number is pushed onto the stack and the execution of the main process is continued recursively (Line 24).

If the user enters an operator the specification checks which operator has been entered. If the `display` operator has been selected and the stack is not empty (Line 9) then the specification calls the `DisplayStack` process and the `Main` process subsequently without changing the stack (Line 10).

If the entered operator equals `add` and the size of the stack is greater than or equal to two (Line 13) then we recursively continue with a stack where the two top elements are replaced with their sum (Line 14).

If the stack size does not meet the requirements of the entered operator (Lines 17 and 18) then the calculator issues an error and continues without changing the stack (Line 19).

The `DisplayStack` process (Lines 27–33) takes a stack *s* and displays the content of the stack using the `out` gate.

The overall behavior of the calculator's specification (Line 3) is composed of instantiating the main process with an empty stack (i.e., nil) with the possibility to disable operation of the main process at any time using a `quit` command.

2.3. Test purposes and test cases

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 Samstag [11], Tgv [4] and Microsoft's Xrt [12] use test purposes for test generation. We use Tgv in our experiments; the formal notation of test purposes for Tgv is given by:

Definition 5 (*Test purpose*). Given a specification S in the form of an IOLTS, a test purpose is a deterministic IOLTS $TP = \left(Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP}\right)$, equipped with two sets of trap states: $Accept^{TP}$ defines pass verdicts, and $Refuse^{TP}$ allows one to limit the exploration of the graph S. Furthermore, $A^{TP} = A^S$ and TP is complete (i.e., it allows all actions in each state).

Fig. 3 serves to illustrate a test purpose and the used notation for test purposes within this paper. We use '$*$'-labeled edges to denote all edges that are not explicitly specified for a particular state. Edge labels that end with '.$*$' denote any edge that start with the text stated before '.$*$'. The illustrated test purpose is intended for our stack calculator specification of Fig. 2. This test purpose selects traces of the specification's LTS that end with $\langle$`out!2, out.*`$\rangle$. It refuses to select any trace that contains `out!error`.

According to Jard and Jéron [4] test synthesis within Tgv is conducted as follows: given a test purpose TP and a specification S Tgv calculates the synchronous product $SP = S \times TP$. The construction of SP is stopped in *Accept* and *Refuse* states as subsequent behaviors are not relevant to the test purpose. Then Tgv extracts SP^{VIS} of SP by removing all τ actions from SP, by adding suspension labels to SP and by making SP deterministic. SP^{VIS} denotes the visible behavior of SP, i.e. SP^{VIS} does not comprise τ labeled (unobservable) transitions and SP^{VIS} is deterministic.

The added suspension labels are needed for input output conformance testing and indicate the absence of output. A suspension label is a δ labeled transition, which indicates the absence of outputs. Thus, δ labeled transitions are enabled in states where neither an output nor a τ labeled transition is enabled.

A test case derived by Tgv is controllable, i.e., it does not have to choose between sending different stimuli or between waiting for responses and sending stimuli. This is achieved by selecting traces from SP^{VIS} that lead to *Accept* states and pruning edges that violate the controllability property. Finally, the states of the test case are annotated with the verdicts pass, fail and inconclusive (inconc).

```
1  specification StackCalc [ui,out]:exit
2  behavior
3    Main[ui,out](nil) [> ui !quit; exit
4  where
5    process Main[ui, out](s:Stack):noexit:=
6    (
7     ui ? op : Operator;
8     (
9      [(op eq display) and (size(s) gt 0)]→(
10       DisplayStack[out](s) >> Main[ui,out](s)
11     )
12     []
13     [(op eq add) and (size(s) ge Succ(Succ(0)))]→(
14       Main[ui,out](push(top(s)+top(pop(s)), pop(pop(s))))
15     )
16     []
17     [((op eq add) and (size(s) lt Succ(Succ(0))))
18      or ((op eq display) and (size(s) eq 0))]→ (
19       out !error; Main[ui,out](s)
20     )
21    )
22   )
23   []
24   ( ui ? num: Nat; Main[ui,out](push(num,s)) )
25   endproc
26
27   process DisplayStack[out](s:Stack):exit:=
28    [size(s) eq Succ(0)] →
29      out !top(s); exit
30    []
31    [size(s) gt Succ(0)] →
32      out !top(s); DisplayStack[out](pop(s))
33   endproc
34 endspec
```

Fig. 2. Lotos specification of a simple stack calculator.

Inconclusive verdicts denote that neither a pass nor a fail verdict has been reached but the implementation has chosen a trace that is not included in the traces selected by the test purpose.

As a major strength of TGV, the test case synthesis is conducted on-the-fly: parts of S, SP, and SP^{VIS} are constructed only when needed. In practice, this allows one to apply TGV to large specifications.

The constructed test case is again an IOLTS with special properties. More formally,

Definition 6 (*Test case*). 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 satisfies the following properties:

1. Inputs of TC are outputs of the implementation under test (IUT) and vice versa. Furthermore, TC considers all possible outputs of the IUT.
2. From each state a verdict must be reachable.
3. States in *Fail* and *Inconc* are only directly reachable by inputs.
4. A test case is input complete in all states where an input is possible.
5. TC is controllable, i.e., no choice between two outputs or between inputs and outputs.

A *test suite* is a set of test cases.

3. Coverage based test purpose generation

One of the main difficulties in testing software is to decide which test cases out of a possibly infinite set to choose. Coverage criteria are a well-known method to focus on finite subsets that exercise some defined aspects of a system. Code coverage criteria are mostly used to assess the quality of an existing test suite, while criteria based on specifications and models are often used to derive test suites. In general, a coverage criterion defines a set of test requirements that should be exercised. The coverage value expresses the percentage of the test requirements that are actually covered by a given test suite.

In this section we define coverage criteria for LOTOS specifications, and show how the LTS representing the semantics of a LOTOS specification has to be extended in order to allow coverage of the defined criteria. Finally, we present a generic set of test purposes which allow one to generate test cases achieving maximum specification coverage.

3.1. Basic coverage of LOTOS specifications

The behavioral part of a LOTOS specification is expressed in terms of processes and their communication. A process is considered to be a black-box at some level of abstraction where only the process' external behavior is considered. Similar to function coverage for sequential programs, process coverage gives the percentage of processes executed within a LOTOS specification.

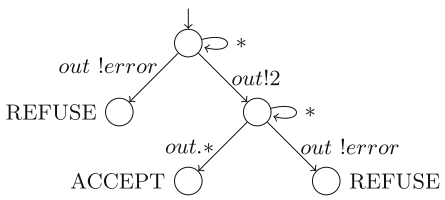

Fig. 3. Example test purpose.

Definition 7 (*Process coverage*). A process of a LOTOS specification is *covered*, if it is executed. The process coverage represents the percentage of processes of the specification that are covered.

The behavior of a LOTOS process consists of actions. The idea of action coverage for LOTOS specifications is that every action statement in the specification should be executed at least once, similar to statement coverage [13] for source code.

Definition 8 (*Action coverage*). An action of a LOTOS specification is *covered*, if it is executed. The action coverage represents the percentage of actions of the specification that are covered.

A single action statement within the specification may generate several edges within the underlying LTS, because data is handled by enumeration. If an action has parameters, e.g., a Boolean variable, then the LTS contains all possible enumeration of the parameters' values. For one Boolean parameter the LTS contains two edges: one labeled with true and another one labeled with false. Action coverage only requires that one of these edges is taken by a test case.

For example, the specification illustrated in Fig. 2 has actions in Lines 7, 19, 24, 29, and 32. The test case shown in Fig. 4 covers 66.6% of these actions: The action in Line 24 is covered by the first two transitions `ui!1` and `ui!2`. The transition `ui!display` covers the action of Line 7. The final two transitions `out!2` and `out!1` cover the two actions of Line 32 and of Line 29, respectively. The process coverage of this test case is 100.0% because every process (`Main` and `DisplayStack`) has been entered at least once. Note that in this example the edges leading to fail states do not add to action nor to process coverage, because there is no test run on our specification that leads to a fail verdict state.

3.2. Logical coverage criteria for LOTOS specifications

LOTOS specifications contain Boolean expressions within guards. For example, Line 9 in Fig. 2 contains the logical expression `(op eq display) and (size(s) gt 0)`. Following a large body of literature of coverage in software testing, we call such an expression a *decision*. A decision comprises a number of *conditions* (also known as *clauses*), that are connected by logical operators. In this section we adapt several known coverage criteria for logical expressions to LOTOS specifications.

Definition 9 (*Decision coverage*). A decision is covered, if it evaluates to true and to false at some point during test execution. The decision coverage represents the percentage of decisions in a LOTOS specification that are covered.

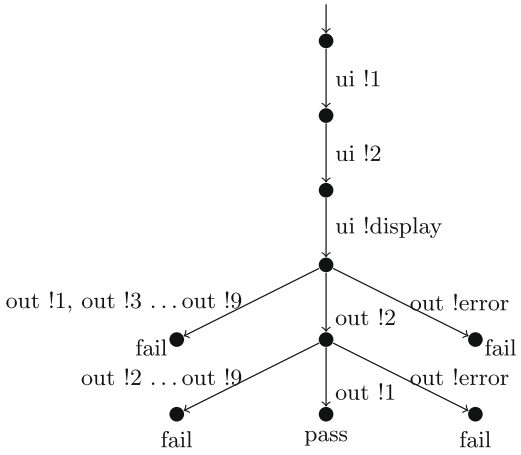

Fig. 4. Test case having 100.0% process coverage and 66.6% action coverage on the specification of Fig. 2.

Decision coverage has also been called *branch coverage* [13] or *predicate coverage* [14]. In contrast to action coverage, each decision results in two test requirements, one for the decision to evaluate to true and one for it to evaluate to false; this applies to the condition and to the condition/decision criterion as well. As an example, the specification given in Fig. 2 has five decisions on Lines 9, 13, 17, 28, and 31. The test case of Fig. 4 achieves 70% decision coverage with regard to these decisions: The transition `ui!display` and the fact that the stack size is changed by the two predecessor transitions cause the guard of Line 9 to evaluate to true and the guards of Line 13 and of Line 17 to evaluate to false. The last two transitions `out!2` and `out!1` cause the decisions of Line 28 and Line 31 to evaluate to true and to false, respectively. The transitions `ui!1` and `ui!2` do not directly trigger any guard.

A decision consists of conditions separated by logical operators (e.g. and, or); these conditions are considered by condition coverage (also known as clause coverage [14]).

Definition 10 (*Condition coverage*). A decision consists of conditions separated by logical operators (e.g. and, or). A single condition is covered, if it evaluates to both true and false at some point during test execution. The condition coverage represents the percentage of conditions in a LOTOS specification that are covered.

For example, the specification listed in Fig. 2 has ten conditions on Lines 9, 13, 17, 28, and 31. The transition `ui!display` of the test case illustrated in Fig. 4 lets the first condition of Line 9 evaluate to true. Because the two transitions `ui!1` and `ui!2` raise the stack size to two, the transition `ui!display` also makes the condition `(size(s) gt 0)` and `(size(s) ge Succ(Succ(0)))` true. Furthermore, the conditions `(op eq add)`, `(size(s) lt Succ(Succ(0)))`, and `(size(s) eq 0)` evaluate to false. The two transitions `out!2` and `out!1` leading to the pass state make the conditions of Lines 28 and 31 evaluate to false and true, and to true and false, respectively. Thus, the condition coverage is 60%.

As satisfying condition coverage does not guarantee that all decisions are covered (i.e., decision coverage is not subsumed by condition coverage), it is common to define the condition/decision (CD) coverage criterion as the combination of decision and condition coverage.

Definition 11 (*Condition/decision coverage*). The condition/decision coverage represents the percentage of conditions and decisions in a LOTOS specification that are covered.

This means that 100% condition/decision coverage is achieved if all conditions evaluate to true and to false, and if every decision also evaluates to true and to false during the test execution.

Our stack calculator specification includes ten conditions and five decisions (Lines 9, 13, 17, 28, 31). Counting the condition/decision coverage of our test case shown in Fig. 4, we get a CD coverage of 63%.

Another coverage criterion for logical expressions is the modified condition/decision coverage (MCDC) [15]. The essence of modified condition/decision coverage is that each condition must show to independently affect the outcome of the decision. That is, test cases must demonstrate that the truth value of the decision has changed because of a change of a single condition.

Definition 12 (*Modified condition/decision coverage*). The modified condition/decision coverage represents the percentage of conditions that independently effected the outcome of a decisions in a LOTOS specification during execution.

For each condition a pair of test cases is needed, i.e., one making the decision true while the other makes the decision false. However, by overlapping these pairs of test cases usually $N+1$ test cases are sufficient for covering a decision comprising N conditions.

Table 2 depicts a minimal set of truth values for the conditions of the guard of Line 17 in Fig. 2. A set of test cases that cause the conditions to take these truth values has 100% MCDC coverage on the guard of Line 17. For example, consider the two test cases 1 and 5. They only differ in the truth value of the condition C_4. However, the truth value of the decision is different for the two test cases 1 and 5 as well. Thus, this test case pair covers C_4 with respect to MCDC coverage. Furthermore, C_3 is covered by the test cases 4 and 5, C_2 is covered by the test cases 1 and 3 and C_1 is covered by the two test cases 2 and 3.

3.3. Weak and strong coverage

Due to the inherent parallel nature of LOTOS specifications we need to distinguish between weak and strong coverage. At any point of execution one may choose between different actions offered by a LOTOS specifications. Obviously, the offered actions depend on the structure of the specification.

For example, consider a specification having two processes $P1$ and $P2$ in parallel, i.e., $P1|||P2$. The $|||$ parallel execution operator denotes any interleaving of the actions offered by $P1$ and $P2$. Thus, at the very beginning of this specification fragment one can choose between the actions initially offered by $P1$ and the actions initially offered by $P2$. A similar situation can be constructed for guarded expressions. An action may be offered only if a particular guard evaluated to true.

A coverage item is covered weakly if the actions related to that coverage item are offered by the specification at some moment during the execution of a test case. Weak coverage does not require that a test case synchronizes on the relevant actions. More formally, weak coverage is defined as follows:

Definition 13 (*Weak coverage*). Given a set of actions $C = \{a_1, \ldots, a_n\}$ comprising the actions relevant for a particular coverage item, an IOLTS representing a test case $t = \left(Q^t, A^t, \rightarrow_t, q_0^t\right)$, and a specification $S = \left(Q^S, A^S, \rightarrow_S, q_0^S\right)$, then t weakly covers S with respect to g, iff

$$\exists a \in C \cdot \exists \sigma \in traces(t) \cdot a \in init(S \textbf{ after } \sigma)$$

Depending on the coverage criterion the relevant actions differ. In the case of process coverage the relevant actions C are given by the transition labels of the transitions that are enabled in the LTS because of the process' behavior. For action coverage the relevant

Table 2
Modified condition/decision coverage for the guard `((op eq add) and (size(s) lt Succ(Succ(0)))) or ((op eq display) and (size(s) eq 0))` of Fig. 2.

TC	C_1 `op eq add`	C_2 `size(s) lt Succ(Succ(0))`	C_3 `op eq display`	C_4 `size(s) eq 0`	$(C_1 \wedge C_2) \vee (C_3 \wedge C_4)$
1	T	F	T	F	F
2	F	T	T	F	F
3	T	T	T	F	T
4	T	F	F	T	F
5	T	F	T	T	T

actions C are given by the transition labels that result from the action to be covered. For logical coverage criteria we have to distinguish between the *true* and the *false* case. In one of these cases there are more transitions enabled in the LTS than in the other. Without loss of generality assume that the *true* case has more enabled transitions. Then, C for the *true* case is given by the transition labels that are absent in the *false* case. The actions C for the *false* case are all actions that are enabled in the *false* case.

We say we have strong coverage if the actions of interest are not only offered by the specification, but if the test case synchronizes on one of that actions. That is, if one action is taken by the test case. More precisely, strong coverage for a coverage goal, expressed in terms of actions, is defined as follows:

Definition 14 (*Strong coverage*). Given a set of actions $C = \{a_1, \ldots, a_n\}$ comprising the actions relevant for a particular coverage item, an IOLTS representing a test case $t = \left(Q^t, A^t, \rightarrow_t, q_0^t\right)$, and a specification $S = \left(Q^S, A^S, \rightarrow_S, q_0^S\right)$, then t strongly covers S with respect to g, iff

$$\exists \sigma \in traces(t) \cdot C \cap init(t||S \textbf{ after } \sigma) \neq \emptyset$$

As this definition shows, strong coverage only requires that at least one of the relevant actions is taken by the test case.

The following example serves to illustrate the difference between weak and strong coverage. Fig. 5 depicts the initial part of the underlying labeled transition system of our stack calculator specification (Fig. 2). As this LTS shows, after executing `ui!add` (Line 7) the calculator may either issue an error (`out!error`) or the user may quit (`ui!quit`) the calculator.

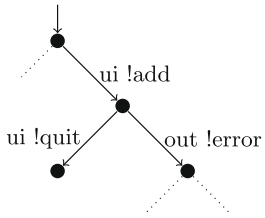

Fig. 5. Part of the underlying LTS of the stack calculator specification.

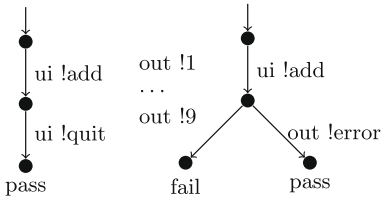

Fig. 6. Two different test cases illustrating weak and strong coverage.

For this part of the specification two different valid test cases are possible; these test cases are illustrated in Fig. 6. Both test cases enable the block guarded by the condition in Line 17. However, the test case on the left hand side does not take an action from the guarded block, but chooses the quit action. This test case covers the decision's true case of the guard weakly. On the other hand, the test case illustrated on the right of Fig. 6 chooses an action that is possible only because the guard has evaluated to true. Thus, this test case achieves strong coverage with respect to the conditions and decisions of interest.

3.4. Relation between coverage criteria

Table 3 relates the coverage criteria of Section 3.1 and of Section 3.2 with respect to weak (W) and strong (S) coverage. The table is read from left to top. For example, the intersection of the first row with the second column states that weak process coverage (Process (W)) is not subsumed by the strong process coverage (Process (S)).

As this table shows, neither weak nor strong process coverage subsume any other coverage criteria. This is also true for weak and strong action coverage. In particular, action coverage does not subsume process coverage because LOTOS allows one to use processes which do not comprise actions.

Furthermore action coverage neither subsumes any other coverage criterion nor is subsumed by any other coverage criterion. Let a, b, and c be actions and let `a[]b[][x>5]→(c)` be the specification. Then a test suite having 100% action coverage consists at least of three test cases while the test suites for our logical criteria comprise only two test cases.

Weak coverage of any logical criteria is always subsumed by its strong pedant. Contrary, weak coverage never subsumes strong coverage. This follows from the definitions of weak and strong coverage. Thus, weak condition decision coverage does not subsume strong decision coverage. However, weak CD coverage subsumes weak decision coverage and strong CD coverage subsumes strong decision coverage.

Strong MCDC subsumes all of our coverage criteria except process and action coverage. This is, because for each condition MCDC comprises one test case such that condition evaluations to both true and false. In addition, there is at least one test case making the decision true and at least one test case making the decision false.

3.5. Deriving test purposes from coverage criteria

We have defined the coverage criteria on the syntactical level of the specification while test purposes operate on the semantic level,

Table 3
Subsumption relation of the approached coverage criteria.

	Process		Action		Decision		Condition		CD		MCDC	
	(W)	(S)	(W)	(S)	(W)	(S)	(W)	(S)	(W)	(S)	(W)	(S)
Process (W)		$\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$
Process (S)	$\not\subseteq$		$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$
Action (W)	$\not\subseteq$	$\not\subseteq$		$\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$
Action (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$
Decision (W)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\subseteq$	$\not\subseteq$	$\not\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$
Decision (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\subseteq$	$\not\subseteq$	$\subseteq$
Condition (W)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$
Condition (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\not\subseteq$	$\subseteq$	$\not\subseteq$	$\subseteq$
CD (W)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\subseteq$	$\subseteq$	$\subseteq$
CD (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\not\subseteq$	$\subseteq$
MCDC (W)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$		$\subseteq$
MCDC (S)	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	$\not\subseteq$	

e.g., on the labeled transition system. Unfortunately, the LTS does not explicitly contain the information needed to generate coverage based test purposes. In order to bridge this gap we have to annotate the LOTOS specification with additional information such that this information is also visible in the LTS.

More specifically, we insert probes α that are not in the initial alphabet of the specification S, i.e., $\alpha \notin A^S \cup \{\tau\}$; we call a specification with inserted probes *marked*. Each probe results in a distinct transition in the labeled transition system, and therefore makes coverage relevant information of the specification visible. By selecting α labeled transition in the test purpose we can now derive test purposes with respect to a certain coverage criterion. Note that for the final test case the α label has to be hidden again.

In the following we explain for every coverage criterion what probes are needed and what the test purposes look like.

Process coverage: for process coverage we generate as many marked specification as there are processes within the specification. Each marked specification comprises a single probe which is inserted directly after the process declaration. In the case of our stack calculator example we generate 2 different copies. In one copy there is an α-probe at Line 6, while in the other copy the α-probe is inserted at Line 28 (in front of the guard).

By using the test purpose depicted in Fig. 7 we derive test cases for each of the specifications in order to obtain a test suite for process coverage. This test purpose simply states that any sequence covering an α-labeled transition followed by an arbitrary action is a valid test case. (Alternatively, one could use a single specification with several probes without any loss of generality. However, our experiments have shown that this slows down TGV significantly.)

Action coverage: in order to generate test cases with respect to action coverage we construct as many marked specifications as there are actions within the specification. For example, for our stack calculator specification we generate 5 different copies. In each copy we insert an α after the action of interest, e.g., after the action statements on Lines 7, 19, 24, 29, and 32. Then, the test purpose depicted in Fig. 8 is used with each of the specifications in order to derive test cases with respect to action coverage. This test purpose simply states that any sequence covering a transition labeled with α is a valid test case; because α follows after the considered action, this guarantees that the action was executed.

Decision coverage: for generating test cases with respect to logical coverage criteria we have to equip our α probes with additional information, which allows us to select a certain outcome of a decision. In the case of decision coverage we simply add the whole decision to the probe.

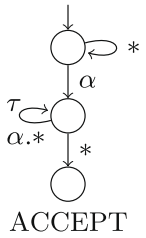

Fig. 7. Test purpose for generating test cases with respect to process coverage of marked specifications.

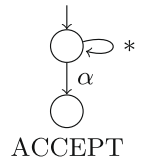

Fig. 8. Test purpose for generating test cases with respect to action coverage of marked specifications.

Fig. 9 serves to illustrate this technique. This figure shows the DisplayStack-process of our stack calculator specification. In order to generate test cases for covering the condition of Line 4, we added a probe equipped with the decision in Line 2. The underlying LTS of this piece of specification contains the edges α!TRUE and α!FALSE, which can be selected directly by the test purpose.

Note that we insert the probes before the decisions if they are not part of a choice operator (i.e., "[]"). In the case of choices we insert the marker before the first condition of the choice (e.g., the probe for the condition of Line 6 in Fig. 9 is inserted at Line 2). For each marked specification we get two test cases from the two test purposes of Fig. 10, one for each possible outcome of a decision. In order to ensure that the decision is evaluated we need to select one non-internal edge, e.g., an edge not labeled with τ. Therefore, we have an '$*$'-labeled edge leading to the accept state, which selects an edge which is possible after α.

Condition coverage: in order to generate test cases with respect to condition coverage we insert our α probes in the same manner as for decision coverage. Instead of equipping the probes with the whole decision we split the decision into its conditions. For example, the probe for covering the two conditions in the guard in Line 9 of our stack calculator example is α `!(op eq display) !(size(s) gt 0)`. The corresponding test purposes are illustrated in Fig. 11. For each condition of our inserted markers (α) we need to select the true (!TRUE) and the false (!FALSE) outcome while we do not care about the other conditions (![A-Z]$*$). Since we need to select a particular condition we have to add ![A-Z]$*$ at the right position for every other condition in the label of the test purpose.

If there are n conditions in a decision we need $2 \times n$ test purposes in order to derive test cases with respect to decision coverage. As for decision coverage we need to select one edge after the α probe in order to ensure that the condition is evaluated.

Condition/decision coverage: we generate test cases with respect to CD coverage by inserting probes similar to condition coverage. The probes are equipped with the decision as the first element followed by the conditions of the decision. We then use the test purposes of Fig. 11 and instantiate them for all parameters of the probe, i.e., for the decision and for all conditions.

Modified condition/decision coverage: the probes inserted into specifications for deriving MCDC based test purposes are similar to the probes for the other logical coverage criteria. Contrary, the test purposes do not contain "![A–Z]"-elements which match any logical value, e.g., true and false, but each label of a test purpose matches a particular Boolean value of every condition. There are several possibilities to calculate truth values for conditions that result in MCDC pairs; we refer to Ammann and Offutt [16] for an overview. In order to generate a minimal number of test purposes for a decision we calculate the truth-table for that decision. Then we search for all sets of truth-value assignments to the conditions of the decision such that each set is a valid MCDC test suite of minimal length. From these sets we randomly select one set which then forms our test suite. The truth-value assignments are directly used within the test purposes. In total, we generate $N + 1$ different test purposes for N conditions. Each test purpose selects a particular valuation of the conditions and of the decision of a probe. For example, let Table 2 be the selected set of truth-value assignments for the decision `((op eq add) and (size(s) lt Succ (Succ(0)))) or ((op eq display) and (size(s) eq 0))` in Line 17 of Fig. 2. Then the generated test purposes for C_1 and C_2 of Table 2 look like those illustrated in Fig. 12. We omit the test purposes for C_3 and C_4 here.

Our current implementation does not consider whether the selected truth-value assignments are valid within our specification or

```
1 process DisplayStack[out,α](s:Stack):exit:=
2  α !(size(s) eq Succ(0));
3  (
4    [size(s) eq Succ(0)] -> (out !top(s); exit)
5    []
6    [size(s) gt Succ(0)] ->
7      (out !top(s); DisplayStack[out](pop(s)))
8  )
9 endproc
```

Fig. 9. Stack calculator specification with probe to cover a condition.

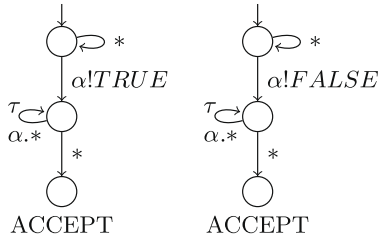

Fig. 10. Test purposes for generating test cases with respect to decision coverage of marked specifications.

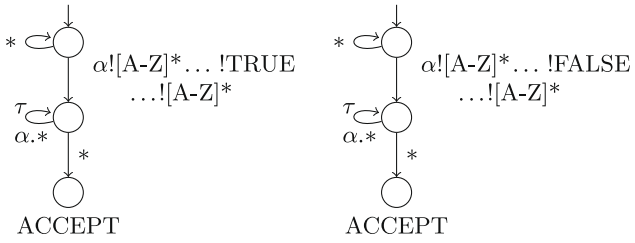

Fig. 11. Test purposes for generating test cases with respect to condition coverage of marked specifications.

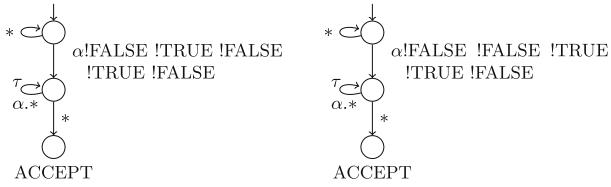

Fig. 12. Test purposes for generating test cases with respect to modified condition/decision coverage of marked specifications.

not. Thus, it may happen that we use truth-value assignments which can never occur within our specification.

3.5.1. Remarks

Except for action coverage, our test purposes can guarantee weak coverage only. A solution for achieving strong coverage would be to insert other probes, e.g., β where $\beta \notin A^S \cup \{\tau\}$, after the actions relevant for a particular coverage item. For example, for process coverage one could insert a β after every action than can occur as the first action within a process.

However, while we can always decide where to insert regular α probes based on the syntax of the specification, this cannot be done on the syntactical level for β probes. We may need to evaluate parts of the specification before we can decide on the positions of β probes. However, if such probes are inserted into the specification, test purposes can make use of these β actions in order to ensure that the right branch within the LTS is taken.

For example, suppose we want to strongly cover a process that does not comprise any action but calls other processes. Further suppose, that these other processes do not return but call themselves recursively. In that case we need to insert β after the first action of the called processes.

3.6. Supporting manually designed test purposes

Our probe insertion technique does not only allow one to automatically generate test suites based on coverage criteria. It may also be used to complement manually designed test purposes.

Test cases generated for manually designed test purposes may not fully cover all aspects of a particular model. By the use of our probe-based technique we can determine the coverage value of a given set of test cases. Furthermore, we can automatically generate test cases that exercise parts of the specification missed by the test cases derived from manually designed test purposes.

More precisely, we do the following in order to complement test cases generated for manually designed test purposes. Given a test suite derived from a set of test purposes we run the test cases of that test suite on a specification comprising all probes for a particular coverage criterion. From these runs we can extract the set of covered probes. By generating test cases for the uncovered probes we can increase the overall coverage value of the test suite.

For example, suppose we want to complement the manually designed test purpose illustrated in Fig. 3 with respect to action coverage for our stack calculator specification. Assuming that the test case generated by TGV for this test purpose looks like the test case depicted in Fig. 4, running it on a copy of the specification comprising all α probes for action coverage gives a coverage value of 66.6%.

From this test run we can deduce that the test case misses the probes corresponding to the action `ui!quit` in Line 3 and to the action `out!error` in Line 19. We can automatically generate two test purposes, which aim to cover the two missed probes. These two test purposes will lead to test cases similar to the test cases illustrated in Fig. 6. The complemented test suite comprises three test cases, which in total achieve 100% action coverage on the specification.

3.7. Reduction

In practice there is often insufficient time for thorough testing activities within industrial projects. Therefore it is reasonable to try to reduce the size of generated test suites. However, the effect of the reduction on the fault-detection ability of the test suites should be small.

The techniques proposed in this paper can be used to apply reduction during test case generation. A single test case may cover more than the coverage item it has been generated for. When using a probe based technique as described in this paper it is easy to identify all items covered by a particular test case. This is done by running the generated test case on a specification containing all probes.

Table 4
Details of the used LOTOS specifications.

	Number of processes	Number of actions	Number of decision	Number of conditions	Number of datatypes	Net lines of code	
						Total	Datatypes
SIP	10	27	39	49	20	3000	2500
CP	16	26	32	39	1	900	700

An optimal test suite is a minimal set of test cases such that all items are covered. Unfortunately, this is equivalent to the set covering problem which is known to be NP-complete. We can, however, approximate the optimal test suite. After a test case has been generated we run this test case on a specification containing all probes and extract the covered probes. We skip test purposes for probes that are covered by previously generated test cases. Note that this minimizes both the number of test cases and the number of invocations of TGV. The number of generated test cases depends on the processing order of the different test purposes.

Note that there is still some potential for reducing the test suite sizes left. We currently do not consider that test cases may cover probes of previously generated test cases. This would possibly allow to remove some more test cases from the generated test suites.

4. Experimental results

For an evaluation of the presented methods we show the results obtained when applying our approach to two different protocols. For both protocols we derived test suites based on the coverage criteria presented in this paper. We show the results obtained when running the generated test suites on real implementations of the two protocols. Furthermore, we present results for complementing manually designed test purposes.

4.1. Applications under test

For our empirical evaluation we used two different applications: (1) Session Initiation Protocol (SIP) [17] and (2) Conference Protocol (CP) [18]. Table 4 summarizes the characteristics of the two specifications in terms of the number of processes, the number of actions, the number of decisions and conditions and in terms of net lines of code.

4.1.1. The SIP registrar application

The Session Initiation Protocol (SIP) [17] handles communication sessions between two end points. The focus of SIP is the signaling part of a communication session independent of the used media type between two end points. More precisely, SIP provides communication mechanisms for *user management* and for *session management*.

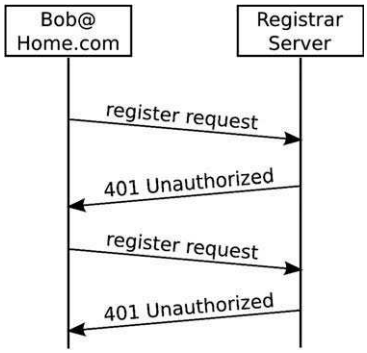

Fig. 13. Simple call-flow of the registration process.

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 *Registrar*, which is responsible for maintaining location information of users.

An example call flow of the registration process is shown in Fig. 13. In this example, Bob tries to register his current device as end point for his address Bob@home.com. Because the server needs authentication, it returns "401 Unauthorized". This message contains a digest which must be used to re-send the register request. The second request contains the digest and the user's authentication credentials, and the Registrar accepts it and answers with "200 OK". For a full description of SIP we refer to the RFC [17].

We developed a formal specification covering the full functionality of a SIP Registrar. 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 a technical report [19].

We used two different implementations of the Session Initiation Protocol Registrar for our evaluation: A commercially available implementation, and version 1.1.0 of the open source implementation OpenSER.[1] Note that a Registrar does not necessarily need to authenticate users. According to the RFC a SIP Registrar SHOULD authenticate the user. This is reflected in our specification by having a non-deterministic internal choice between using authentication and an unauthenticated mode. As both tested implementations allow one to turn authentication off, we run all our test cases against two different configurations (with and without authentication).

4.1.2. Conference Protocol

The Conference Protocol has been used previously to analyze the fault-detection ability of different formal testing approaches (e.g., [2,20]). The specification is available in different specification languages to the public.[2]. In addition, there are 27 erroneous implementations which can be used to evaluate testing techniques. Each faulty implementation comprises a single fault.

The protocol itself is a simple communication protocol for a chat application. The main part of the application is called the Conference Protocol Entity (CPE). A CPE serves as chat client with two interfaces; one interface allows the user to enter commands and to receive messages sent by other users, and the other interface allows the chat application to send and receive messages via the network layer. These two interfaces are the points of control and observation for a tester.

Users can join conferences, exchange messages and leave conferences. Each user has a nick name and can only join one conference at a time. Fig. 14 shows a typical example of a simple chat session. First, user Bob joins conference "C1" using the nickname Bob. The Conference Protocol entity sends that information to all potential conference partners. In the illustrated scenario user Alice participates in the same conference as joined by Bob. Thus, Alice's

[1] http://www.openser.org.
[2] http://fmt.cs.utwente.nl/ConfCase/.

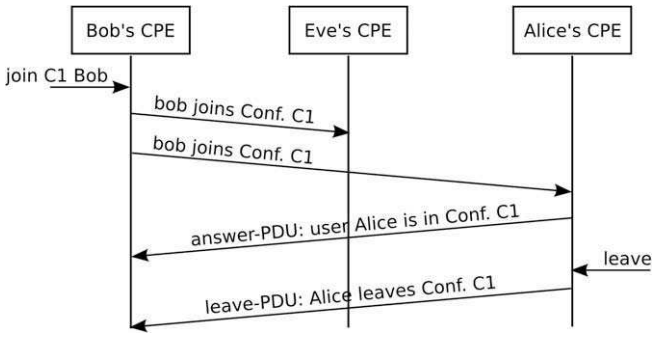

Fig. 14. Simple call-flow illustrating joining and leaving a chat session.

protocol entity answers with an *answer*-protocol data unit (PDU). Then Alice decides to leave the conference which causes her protocol entity to send a *leave*-PDU to all participating users, i.e., to Bob's CPE.

4.2. Test case generation results

We have implemented a tool that takes a LOTOS specification, automatically generates probe annotated specifications for the coverage criteria presented in Section 3, and instantiates our generic test purposes for the inserted probes.

For the SIP Registrar specification our tool generated 10 probe annotated specifications for process coverage (P), 27 probe annotated specifications for action coverage (A), and 39 probe annotated specifications for condition coverage (C), decision coverage (D), condition/decision (CD) coverage and modified condition/decision coverage (MCDC), respectively. For decision, condition, condition/decision, and for modified condition/decision coverage the number of generated test purposes does not only depend on the number of inserted probes but also on the number of conditions within a decision. For those coverage criteria our tool generated 78 (D), 98 (C), 176 (CD), and 94 (MCDC) test purposes, respectively.

For the Conference Protocol specification our tool derived 16 probe annotated specifications for process coverage, 26 probe annotated specifications for action coverage and 32 probe annotated specifications for each of the logical coverage criteria. The tool generated 64 (D), 78 (C), 142 (CD), and 71 (MCDC) test purposes, respectively.

These generated test purposes together with the annotated specifications serve as basis for our experimental evaluation.

4.2.1. Generating test cases based on coverage criteria only

Tables 5 and 6 list the results when generating test cases for the coverage criterion based test purposes (1st column). These tables show the number of instantiated test purposes (2nd column), the number of generated test cases (3rd and 7th column), the number of test purposes where TGV failed to generate a test case (4th and 9th column) and the time needed to process all test purposes (5th and 10th column). We conducted all experiments on a PC with Athlon(tm) 64 X2 Dual Core Processor 4200+ and 2GB RAM.

TGV fails on some test purposes, because its depth first search algorithm runs into a path not leading to an accept state within the synchronous product between the specification and the test purpose. As our specifications have infinite state spaces and our test purposes lack *Refuse* states TGV eventually runs out of memory. Furthermore, we may generate test purposes for which no test case can be derived, e.g. if there are conditions/decisions that are tautologies or condition/decisions that are unsatisfiable. Also for MCDC we may generate test purposes for which no test case exists (see Section 3.5). However, we did not observe invalid test purposes in our experiments.

The left part of the table depicts the figures obtained when generating test cases for all test purposes. In contrast, the right part illustrates the results gained from reducing the test suites as described in Section 3.7. An additional column (8th column) in this part lists the number of probes additionally covered by the generated test cases.

TGV runs out of memory on 61 (73) test purposes for the SIP (Conference Protocol) application (see 4th column). Thus, for our Session Initiation Protocol we get test suites having 100% process coverage, 92.6% action coverage, 92.3% decision coverage, 95.9% condition coverage, 94.3% CD coverage, and 58.5% MCDC coverage. For the Conference Protocol the generated test suites have 93.8%

Table 5
Test case generation results for the Session Initiation Protocol.

C.	No.	Regular				Reduced				
	TP	Ok	∞	Time	Coverage	Ok	Coverage	∞	Time	Coverage
P	10	10	0	09 m	100.0	2	8	0	08 m	100.0
A	27	25	2	2 h 49 m	92.6	10	15	2	2 h 37 m	92.6
D	78	72	6	8 h 28 m	92.3	10	62	6	3 h 11 m	92.3
C	98	94	4	11 h 13 m	95.9	12	82	4	3 h 26 m	95.9
CD	176	166	10	19 h 39 m	94.3	12	154	10	4 h 30 m	94.3
MCDC	94	55	39	10 h 26 m	58.5	32	26	36	9 h 44 m	61.7
Σ	483	422	61	2 d 04 h 44 m		78	347	58	23 h 36 m	

Table 6
Test case generation results for the Conference Protocol.

C.	No.	Regular				Reduced				
	TP	Ok	∞	Time	Coverage	Ok	Coverage	∞	Time	Coverage
P	16	15	1	6 h 56 m	93.8	3	12	1	6 h 56 m	93.8
A	26	19	7	2 d 03 h 16 m	73.1	12	7	7	2 d 03 h 13 m	73.1
D	64	56	8	4 d 17 h 05 m	87.5	37	20	7	1 d 18 h 55 m	89.1
C	78	66	12	5 d 21 h 38 m	84.6	46	21	11	2 d 11 h 29 m	85.9
CD	142	122	20	5 d 11 h 33 m	85.9	78	46	18	4 d 06 h 20 m	87.3
MCDC	71	46	25	7 d 16 h 40 m	64.8	25	22	24	5 d 07 h 20 m	66.2
Σ	397	324	73	26 d 05 h 08 m		201	128	68	16 d 06 h 13 m	

process coverage, 73.1% action coverage, 87.5% decision coverage, 84.6% condition coverage, 85.9% CD coverage, and 64.8% MCDC coverage, respectively.

When generating test cases for uncovered test purposes only (see Section 3.7) we are sometimes able to cover probes for which TGV runs out of memory otherwise. This is because TGV may run out of memory before the branch containing the probe can be selected during the depth first search. For example, for the SIP application the modified condition/decision coverage increases from 58.5% to 61.7%.

However, as guaranteed by our minimization approach there is never a reduction of model coverage, w.r.t. the coverage criterion, although the overall number of test cases has been reduced by 82% and by 38% for the SIP application and the Conference Protocol Application, respectively.

The test case generation takes much longer for the Conference Protocol than for the Session Initiation Protocol. Fig. 15 serves to illustrate this effect. As this figure shows if TGV is able to generate a test case for a particular test purpose this is usually quite fast, i.e., on average it takes approximately seven minutes for the SIP application and approximately 6 s for the Conference Protocol application. However, if TGV fails to generate a test case for a particular test purpose it takes a long period of time (SIP: on average eight minutes, Conference Protocol: on average 6 h) before it runs out of memory. In particular, for the Conference Protocol it sometimes takes days before TGV runs out of memory.

4.2.2. Supplementing manually designed test purposes

This section comprises the results when combining manually designed test purposes with coverage based test purposes (see Section 3.6). TGV derives one test case for every test purpose.

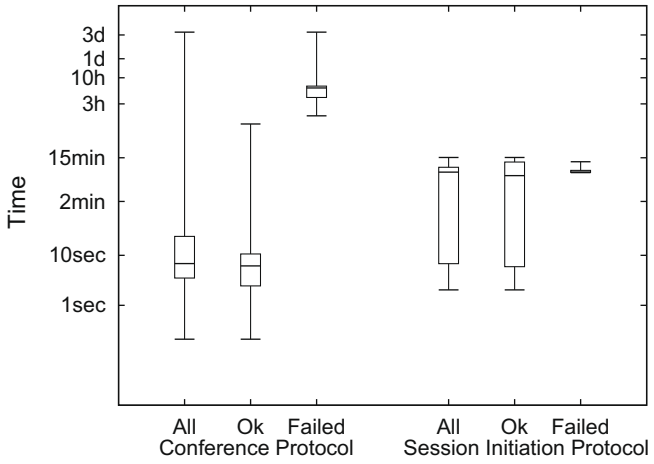

Fig. 15. Minimum, maximum, 1st, 2nd, and 3rd quartile of the test case generation times for the two different protocols (with a logarithmic scaled *y*-axis).

For the Session Initiation Protocol we identified five relevant scenarios from the textual specification and formalized these scenarios in terms of five different test purposes. For the LOTOS version of the Conference Protocol specification we manually developed 10 test purposes. Thus, starting with the manually designed test purposes for the Session Initiation Protocol and for the Conference Protocol we generated five and ten test cases, respectively.

Table 7 contains information about the test suites generated when combining coverage based test purposes with hand-crafted test purposes. It shows for each coverage criterion (1st column) the number of test cases (2nd column and 7th column) contained in the initial test suite, i.e., the test suite directly derived from the hand-crafted test purposes. The 3rd and the 8th column list the coverage value for these initial test suites. The number of missed probes is given by the sum of the successive two columns (i.e., 4th and 5th; 9th and 10th). In these columns the table comprises the number of coverage based test purposes for which TGV succeeded to generate test cases (4th and 9th column) and the number of test purposes for which TGV runs out of memory (5th and 10th column). Finally, in the columns six and eleven the table lists the new coverage values for the extended test suites.

An important insight is that the coverage of the test cases derived from hand-crafted test purposes is not satisfactory. Although we tried to derive the test purposes by identifying relevant scenarios from the informal specifications we missed some parts of the model. For example, the test purposes designed for the Session Initiation Protocol failed to cover a whole process.

By complementing the test purpose using the presented technique we can increase the coverage for both specifications. On average we were able to increase the coverage for the Session Initiation Protocol by 36% and by 14% for the Conference Protocol.

4.2.3. Test case execution

The obtained test cases are abstract test cases, i.e. they are given in terms of labeled transition systems. Such test cases cannot be directly applied to a system under test, since the labels on the transitions are usually not understood by the implementation under test.

To adapt abstract test cases to the IUT it is necessary to reinsert the information abstracted within the model into test case. More precisely, every stimuli i has to be converted to a concrete message $\gamma(i)$, while a system response o has to be mapped to the abstract level $\alpha(o)$ [21].

For executing test cases we have implemented a test case execution framework [22]. This execution framework communicates with implementations via UDP and uses a rule based rewriting system to convert abstract messages to concrete ones and vice versa.

For our SIP Registrar specification [19], the rule sets for γ and α comprise 66 and 36 rules, respectively. Contrary, in the case of the Conference Protocol γ comprises 28 rules, while α consists of 33 rules.

Table 7
Complementing manually generated test cases.

Coverage criterion	Session Initiation Protocol					Conference Protocol				
	No. TC	Coverage (%)	No. TC		New coverage (%)	No. TC	Coverage (%)	No. TC		New coverage (%)
			Ok	∞				Ok	∞	
P	5	90.0	1	0	100.0	10	100.0	0	0	100.0
A	5	66.7	8	1	96.3	10	96.2	0	1	96.2
D	5	59.0	28	4	94.9	10	90.6	6	0	100.0
C	5	52.0	45	2	98.0	10	74.4	19	1	98.7
CD	5	31.8	112	8	95.5	10	50.7	62	8	94.4
MCDC	5	30.9	26	39	58.5	10	74.7	3	15	78.9

Fig. 16 shows an abstract label of a SIP test case and the corresponding concrete message that is obtained by applying the rewriting rules. The rewritten message is sent to the implementation. If there is a response from the implementation, than this response is rewritten to the corresponding abstract message. The abstract message is matched with the labels of the test case. This procedure continues until a verdict state is reached.

By running the test execution framework and the implementation under test on the same host we ensure that messages are delivered in order, even when UDP is used as underlying communication service.

4.2.4. Test execution results

Table 8 shows the results obtained when executing the test cases generated for a certain coverage criterion (1st column) against our two SIP Registrar implementations in terms of the issued verdicts passed (2nd and 6th column), failed (3rd and 7th column), and inconclusive (4th and 8th column) for the OpenSER Registrar (2nd–5th column), and the commercial Registrar (6th–9th column). Furthermore, this table shows the number of differences between the implementations and the specification (5th and 9th column). Note that for the fifth column and for the ninth column the last row (Σ) shows the total number of faults detected by the union of all test suites. This is also the case for all faults labeled columns in all successive tables.

We executed our test cases against the implementations using two different configurations covered by our specification (see Section 4.1.1). The values in Table 8 are the sums of the values for runs with the two configurations.

The 122 failed test cases on the OpenSER Registrar implementation detected three different faults, and the 271 failed test cases for the commercial implementation also revealed three different faults. Two faults are common to both implementations. These faults are related to the rejection of registration messages with too brief registration intervals, and to the rejection of messages containing an incorrect combination of message parameters. In addition, the commercial implementation does not reject messages with an invalid sequence number, which is implemented correctly in the OpenSER Registrar. However, the open source implementation allows a user to delete an existing registration with a message having the same sequence number as the message that created the registration; this violates the RFC.

The test cases derived from the action coverage criterion revealed the missing rejection of messages with too brief registration intervals in both implementations. In addition, they detected the incorrect sequence number problem in the commercial implementation. Our test cases for decision, condition and condition/decision coverage revealed all three faults in both implementations under test.

Contrary, our five manually designed test purposes (see Table 11) detected one and two faults in the OpenSER and in the commercial Registrar implementation. In that case the automatically generated test cases using our coverage based approach outperform the manually designed test purposes.

Table 9 illustrates the results when executing the reduced test suites on the two SIP Registrar implementations. Even though our reduction of the test suite size does not affect the coverage value, it is known that the fault sensitivity can be adversely affected. This can be observed in Table 9, where the reduction of the number of test cases within the test suites also reduces the number of detected faults. The test suites based on condition/decision coverage, on decision coverage and on condition coverage missed one fault that has been revealed by their non-reduced counterparts. The action coverage test suite did not detect any failures in the open

```
PIN !NEWREGMSG (1, 1, 1, 1, FALSE, ADDCONTACTINFO (NEWCONTACTINFO (
                ANY_ADDRESS2, 300), NILCONTACTSET), 3600)
```

```
REGISTER sip:mwsoftnet1.ist.dmz SIP/2.0
Via: SIP/2.0/UDP 10.0.0.1:5065;branch=z9hG4bKb5uXo7N
From: Bob <sip:bob@home.com>;tag=oVahxudCk
To: Bob <sip:bob@home.com>
Call-ID: 8DiF5Kk0LKBCpvv5
CSeq: 1 REGISTER
Expires: 3600
Contact: <sip:192.168.0.100:5065>;expires=300
Content-Length: 0
```

Fig. 16. An abstract test message and the corresponding concrete test message for testing the Session Initiation Protocol.

Table 8
Test execution results using the regular test suite on two different SIP Registrars.

C.	OpenSER				Commercial			
	Pass	Fail	Inconclusive	Faults	Pass	Fail	Inconclusive	Faults
P	18	1	1	1	18	1	1	1
A	40	1	9	1	27	15	8	2
D	100	16	28	3	72	44	28	3
C	124	25	39	3	86	65	37	3
CD	224	41	67	3	158	109	65	3
MCDC	45	38	27	3	46	37	27	3
Σ	551	122	171	3	407	271	166	3

Table 9
Test execution results using the reduced test suite on two different SIP Registrars.

C.	OpenSER				Commercial			
	Pass	Fail	Inconclusive	Faults	Pass	Fail	Inconclusive	Faults
P	2	1	1	1	2	1	1	1
A	12	0	8	0	3	10	7	1
D	10	2	8	2	2	10	8	2
C	12	2	10	2	2	12	10	2
CD	11	2	11	2	1	13	10	2
MCDC	2	36	26	3	3	35	26	3
Σ	49	43	64	3	13	81	62	3

source implementation and missed two faults in the commercial implementation.

Table 10 lists the result when executing the generated test cases on the 27 faulty implementations of the Conference Protocol. This table comprises the results for the regular (2nd–5th column) and for the reduced (6th to 9th column) test suites. As this table shows, our coverage based test cases detected 8 of the 27 faulty implementations. We miss faults mainly because we fail to generate test cases for some test purposes. Furthermore, the generated test cases capture similar scenarios of the specification. For example, we never observed the generation of a test purposes for the scenario of users leaving a conference. Thus, we missed all faults that require at least one leave action of a conference user. Contrary, our manually designed test purposes use leave messages and detected five faults (missed by coverage based testing) which occurred only because of the use of leave messages.

As for the SIP test suites, in some cases the test suite reduction also reduced the fault sensitivity for the Conference Protocol test suites.

While the execution of some test cases on the SIP Registrars lead to inconclusive verdicts there are no inconclusive verdicts when testing the Conference Protocol implementations. Due to the structure of the specifications the test cases for Conference Protocol do not comprise any inconclusive verdict state. This is, because TGV generates an inconclusive verdict if either a refuse state or a sink state is reached during test case generation. As we do not have refuse states in our test purposes and as there are no sink states in the Conference Protocol specifications there are no inconclusive verdicts in the derived test cases.

Contrary, the SIP Registrar specification has sink states because we have an upper bound for the message sequence number in our specification. Thus, there are sink states when this upper bound is reached, which leads to inconclusive verdicts during test case generation.

4.2.5. Results for complemented test suites

Tables 11 and 12 show the results when running the complemented test suites on the SIP applications and on the Conference Protocol applications, respectively.

Table 11 shows the number of test cases (2nd column) used for the different test suites. Furthermore, this table depicts the number of passed (3rd and 7th column), the number of failed (4th and 8th column), and the number of inconclusive (5th and 9th column) verdicts given by the test cases. Finally, the sixth and the tenth column list the number of faults detected in the different implementations.

We again run each test suite on the two different configurations of our SIP Registrars, i.e., each test case runs on two different configurations. Therefore, we have twice as many verdicts as test cases. The first line of the table shows the results of only

Table 10
Test execution results using the regular and the reduced test suites on the 27 faulty implementations of the Conference Protocol.

C.	Regular				Reduced			
	Pass	Fail	Inconclusive	Faults	Pass	Fail	Inconclusive	Faults
P	403	2	0	1	80	1	0	1
A	497	16	0	4	309	15	0	4
D	1347	165	0	8	952	47	0	7
C	1609	173	0	8	1116	126	0	7
CD	2956	338	0	8	1886	220	0	8
MCDC	1134	108	0	8	581	94	0	8
Σ	7946	802	0	8	4924	503	0	8

Table 11
Test execution results using the complemented test suites on two different SIP Registrars.

C.	No. TC	OpenSER				Commercial			
		Pass	Fail	Inconclusive	Faults	Pass	Fail	Inconclusive	Faults
Manual TPs	5	5	1	4	1	3	2	5	2
P	6	6	1	5	1	3	3	6	2
A	13	15	1	10	1	5	10	11	3
D	33	23	7	36	2	3	32	31	3
C	50	41	14	45	3	11	51	38	4
CD	117	130	25	79	3	75	93	66	4
MCDC	31	18	10	34	3	3	31	28	4
Σ	255	238	59	213	3	103	222	185	4

Table 12
Test execution results using the complemented test suite on the Conference Protocol implementations.

C.	No. TC	Pass	Fail	Inconclusive	Faults
Manual TPs	10	195	75	0	17
P	10	195	75	0	17
A	10	195	75	0	17
D	16	328	104	0	17
C	29	650	133	0	18
CD	72	1678	266	0	18
MCDC	13	262	89	0	17
Σ	160	3503	817	0	18

running the test cases derived from the manually created test purposes.

Using the complemented test suites we found two faults for each implementation not detected when using test cases derived from hand-crafted test purposes only. However, also the test suite sizes increased. While having ten test runs (five test cases) for our hand-crafted test purposes we have at most 234 test runs (117 test cases) for CD coverage.

Table 12 uses the structure of Table 11 and lists the results obtained when executing the complemented test suite for the Conference Protocol Specification on the 27 faulty implementations.

For the Conference Protocol we detected one more faulty implementation. The test suite size increased from ten test cases for hand-crafted test purposes to at most 72 test cases when using CD coverage for complementation. In the case of condition coverage the test suite size has been tripled while the number of detected faults has been increased by 5%.

5. Related research

Coverage based testing has a long tradition in software engineering; classical books on software testing, e.g., [13], describe different well-known coverage criteria. Coverage criteria for logical expressions [14] lend themselves not only to test suite analysis, but also to test case generation. For example, model checkers have been used to automatically generate test cases that satisfy coverage criteria [23–25]. However, because testing with model checkers is based on the use of linear counterexamples as test cases, model checkers cannot be used to generate test cases for non-deterministic or incomplete specifications, and cannot be used to generate test cases for the ioco theory.

As the ioco theory is formulated over labeled transition system (LTS) one direction of research is to derive test cases directly from LTSs. Various test case selection strategies, such as random test case selection [5], or coverage of states, labels and transitions [26] have been investigated. Contrary, we are looking for coverage of relevant aspects of the high-level Lotos specification. Coverage based test case generation from Lotos specification was also subject to previous research.

van der Schoot and Ural [27] presented a technique for test case generation with respect to the define and use of variables. They use data flow graphs to identify the relevant traces. The feasibility of these traces is then verified by using guided interference rules on the Lotos specification. However, they derive linear test cases and only address test sequence selection. As a labeled transition system does not comprise any symbolic variables, our approach selects not only the test sequence but also the test data.

Cheung and Ren [28] define operational coverage for Lotos specifications, i.e., coverage criteria that aim to reflect the characteristics of Lotos operations. Furthermore, they propose an algorithm that derives an executable test sequence from a given specification with respect to their coverage criteria. Their algorithm is based on a Petri-net representation of the Lotos specification.

Amyot and Logrippo [29] use a probe insertion technique to measure structural coverage of Lotos specifications. Their probe insertion strategy allows one to reason about the coverage of a test suite for a Lotos specification. Furthermore, they present an optimization that reduces the number of needed probes. However, they only considered action coverage. Furthermore, their approach deals with measuring coverage only. They do not consider generating test cases with respect to their inserted probes.

Automatic generation of test purposes has been considered previously. da Silva and Machado [30] derive test purposes from temporal logic properties using a modified model-checking algorithm which extracts examples and counter-examples from a model's state space. By analyzing these traces they obtain the test purposes.

In the work of Henniger et al. [31] test purposes are derived from communicating extended finite state machines by relying on all-nodes coverage on a labeled event structure derived from the EFSMs. Amyot et al. [32] compare three different approaches to derive test purposes from use case maps.

Aichernig and Delgado [33] use mutation techniques for test purpose generation. They generate faulty specifications using mutation operators, and for each faulty specification they derive a distinguishing sequence (if such a sequence exists) that discriminates the original specification from the faulty one. This sequence serves as test purpose.

We extended the mutation technique [33] in order to apply it to industrial sized specifications with huge, possibly infinite, state spaces [34]. We used probes to mark the place of the mutation, thereby extracting the relevant part of the specification and its mutated version. By comparing the relevant parts, i.e., the parts that are affected by the mutation, we are able to deal with large specifications.

5.1. Comparison of test results

The Conference Protocol case study [18] used in this paper was designed to support the comparison of different test case selection strategies. Thus, others have applied various testing techniques to this application. Table 13 gives an overview of the results achieved by different testing techniques.

By the use of *random* test case selection 25 of the 27 erroneous implementations of the conference protocol have been found [20]. The two missed mutants accept data units from any source, i.e. they do not check if the data units come from potential conference partners. As the authors abstracted from this behavior these two mutants are correct with respect to their specification.

Also the Tgv tool has been applied for testing the Conference Protocol [2]. By the use of 19 *manually designed test purposes*, 24 faulty implementations have been found. However, even after 10 h of test purposes design the authors did not manage to generate a test suite that detect the 25th mutant. Because they use the same specification as Belinfante et al. [20], two of the mutants were correct w.r.t the used specification. Unfortunately, the test purposes used by du Bousquet et al. [2] were not available to us. Thus, we developed our own set of test purposes. Our test purposes lead to a test suite which detects 17 mutants (2nd last row of Table 13).

Heerink et al. [35] applied the Philips Automated Conformance Tester (phact) to the Conference Protocol. As Phact relies on extended finite state machines (EFSM) they developed an EFSM specification for the Conference Protocol. By the use of this specification Phact was able to detect 21 of the 27 faulty implementations.

Table 13
Comparison of test results among different test case selection strategies using the Conference Protocol.

Selection strategy	Conference Protocol		SIP Registrar		
	Number of faults	Ref.	OpenSER	Comm.	Ref.
Random	25	[20]	4	5	[38]
Manual test purposes	24	[2]			
PHACT	21	[35]			
Spec explorer	–	[37]			
Coverage on LTS	25	[26]			
Mutation based			4	6	[34]
Coverage on LOTOS	8	Section 4.2.4	3	3	Section 4.2.4
Manual test purposes	17	Section 4.2.5	1	2	Section 4.2.5
Complemented TPs	18	Section 4.2.5	3	4	Section 4.2.5

Microsoft's SPECEXPLORER tool [36] has been applied to the Conference Protocol by Botinčan and Novaković [37]. Unfortunately, they did not use the 27 available faulty implementations but developed 3 mutants by their own. Thus, the obtained results can neither be compared with the other case studies nor with our results.

Coverage based test case selection on the level of the labeled transition system [26] lead to test suites that also detected 25 of the 27 faulty mutants.

As the conference protocol has been tested using other testing techniques, we also applied different test case selection strategies to our SIP Registrar specification. Using mutation based test case selection on the SIP specification [34] revealed 6 faults in the commercial Registrar implementation and 4 faults in the OpenSER implementation [38]. Random test case selection detected 5 faults in the commercial and 4 faults in the OpenSER Registrar. These figures are also summarized in Table 13.

6. Conclusions

A significant effort has been put into the development of formal frameworks and practical tools for test case generation based on labeled transition systems. The results are convincing, but the fact that test purposes have to be manually formulated makes the testing process dependent on the skills of the tester. In this paper we present a set of generic test purposes based on coverage criteria defined for LOTOS specifications. An extension of the mapping to labeled transition systems is necessary to explicitly reference the necessary information. We evaluated our approach using two different protocol specifications.

In addition to using test cases automatically derived for particular coverage criteria we show how our technique can be used to supplement manually designed test purposes. Therefore, we designed test purposes for our two LOTOS specifications based on the informal protocol descriptions. An interesting result of our experiments is that the coverage achieved by manually designed test purposes is potentially insufficient, therefore a complementary coverage based technique is very recommendable. For example, in the case of the Session Initiation Protocol specification we are able to double the number of detected faults.

In this paper we define an elementary set of coverage criteria, which correspond to some of the most commonly used criteria in software testing. It should be straightforward to define further coverage criteria based on logical expressions, such as, for example, multiple condition coverage [13], or other modified condition/decision coverage variants [14]. The test purposes described in this paper only guarantee weak coverage, as long as only a single test case is derived for each test purpose. Achieving strong coverage will be considered in future work. Finally, while we considered LOTOS in this work, our approach can be applied to any specification language with LTS semantics.

Acknowledgements

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

References

[1] G. Fraser, M. Weiglhofer, F. Wotawa, Coverage based testing with test purposes, in: Proceedings of the Eighth International Conference on Quality Software, 2008, pp. 199–208.
[2] L. du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, R.G. de Vries, Formal test automation: the conference protocol with TGV/TORX, in: Proceedings of 13th International Conference on Testing Communicating Systems: Tools and Techniques, IFIP Conference Proceedings, vol. 176, Dordrecht, 2000, pp. 221–228.
[3] ISO, 8807: information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989.
[4] C. Jard, T. Jéron, TGV: theory, principles and algorithms, International Journal on Software Tools for Technology Transfer 7 (4) (2005) 297–315.
[5] J. Tretmans, E. Brinksma, TorX: automated model based testing, in: A. Hartman, K. Dussa-Zieger (Eds.), Proceedings of the First European Conference on Model-Driven Software Engineering, 2003, pp. 13–25.
[6] J. Tretmans, Test generation with inputs, outputs and repetitive quiescence, Software – Concepts and Tools 17 (3) (1996) 103–120.
[7] T. Bolognesi, E. Brinksma, Introduction to the ISO specification language LOTOS, Computer Networks and ISDN Systems 14 (1) (1987) 25–59.
[8] R. Milner, A Calculus of Communicating Systems, vol. 92, Springer, 1980.
[9] H. Ehrig, W. Fey, H. Hansen, ACT ONE – an algebraic specification language with two levels of semantics, in: M. Broy, M. Wirsing (Eds.), Proceedings Second Workshop on Abstract Data Type, 1983.
[10] J. Tretmans, Model based testing with labelled transition systems, in: R. Hierons, J. Bowen, M. Harman (Eds.), Formal Methods and Testing, Lecture Notes in Computer Science, vol. 4949, Springer, 2008, pp. 1–38.
[11] J. Grabowski, D. Hogrefe, R. Nahm, Test case generation with test purpose specification by MSC's, in: Proceedings of the Sixth SDL Forum, 1993, pp. 253–266.
[12] W. Grieskamp, N. Tillmann, C. Campbell, W. Schulte, M. Veanes, Action machines – towards a framework for model composition, exploration and conformance testing based on symbolic computation, in: Proceedings of the International Conference on Software Quality, 2005, pp. 72–79.
[13] G.J. Myers, The Art of Software Testing, second ed., John Wiley & Sons, Inc. 2004 (Revised and updated by Tom Badgett and Todd M. Thomas with Corey Sandler).
[14] P. Ammann, J. Offutt, H. Huang, Coverage criteria for logical expressions, in: Proceedings of the 14th International Symposium on Software Reliability Engineering, vol. 99, Washington, DC, USA, 2003.
[15] J. Chilenski, S. Miller, Applicability of modified condition/decision coverage to software testing, Software Engineering Journal 9 (1994) 193–200.
[16] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, NY, USA, 2008.
[17] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, SIP: Session Initiation Protocol, RFC 3261, IETF, 2002, <http://www.jdrosen.net/papers/rfc3261.txt>.
[18] R. Terpstra, L.F. Pires, L. Heerink, J. Tretmans, Testing theory in practice: a simple experiment, Tech. Rep., University of Twente, The Netherlands, 1996.
[19] M. Weiglhofer, A LOTOS Formalization of SIP, Tech. Rep. SNA-TR-2006-1P1, Competence Network Softnet Austria, Graz, Austria, 2006.

[20] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L.M.G. Feijs, S. Mauw, L. Heerink, Formal test automation: a simple experiment., in: Twelfth International Workshop on Testing Communicating Systems, vol. 147, IFIP Conference Proceedings, Kluwer, 1999, pp. 179–196.
[21] A. Pretschner, J. Philipps, Methodological issues in model-based testing, in: Model-Based Testing of Reactive Systems, 2004, vol. 281–291.
[22] B. Peischl, M. Weiglhofer, F. Wotawa, Executing abstract test cases, in: Model-based Testing Workshop in conjunction with the 37th Annual Congress of the Gesellschaft fuer Informatik, GI, Bremen, Germany, 2007, pp. 421–426.
[23] A. Gargantini, E. Riccobene, ASM-based testing: coverage criteria and automatic test sequence generation, Journal of Universal Computer Science 7 (11) (2001) 1050–1067.
[24] H.S. Hong, I. Lee, O. Sokolsky, H. Ural, A temporal logic based theory of test coverage and generation, in: Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 2280, Springer-Verlag, Gmbh, 2002, pp. 151–161.
[25] S. Rayadurgam, M.P.E. Heimdahl, Coverage based test-case generation using model checkers, in: Proceedings of the Eighth International Conference and Workshop on the Engineering of Computer Based Systems, Washington, DC, 2001, pp. 83–91.
[26] J. Huo, A. Petrenko, Transition covering tests for systems with queues, software testing, Verification and Reliability 19 (1) (2009) 55–83.
[27] H. van der Schoot, H. Ural, Data flow oriented test selection for lotos, Computer Networks and ISDN Systems 27 (7) (1995) 1111–1136.
[28] T. Cheung, S. Ren, Executable test sequences with operational coverage for LOTOS specifications, in: Twelfth Annual International Phoenix Conference on Computers and Communications, 1993, pp. 245–253.
[29] D. Amyot, L. Logrippo, Structural coverage for LOTOS – a probe insertion technique, in: Proceedings of the 13th International Conference on Testing Communicating Systems: Tools and Techniques, Kluwer, BV, 2000, pp. 19–34.
[30] D.A. da Silva, P.D.L. Machado, Towards test purpose generation from CTL properties for reactive systems, Electronic Notes in Theoritical Computer Science 164 (4) (2006) 29–40.
[31] O. Henniger, M. Lu, H. Ural, Automatic Generation of Test Purposes for Testing Distributed Systems, in: A. Petrenko, A. Ulrich (Eds.), Third International Workshop on Formal Approaches to Testing of Software, vol. 2931, Springer, Montreal, Quebec, Canada, 2003, pp. 178–191.
[32] D. Amyot, L. Logrippo, M. Weiss, Generation of test purposes from use case maps, Computer Networks 49 (5) (2005) 643–660.
[33] B.K. Aichernig, C.C. Delgado, From faults via test purposes to test cases: on the fault-based testing of concurrent systems, in: Proceedings of the Ninth International Conference on Fundamental Approaches to Software Engineering, vol. 3922, LNCS, Springer, 2006, pp. 324–338.
[34] B.K. Aichernig, B. Peischl, M. Weiglhofer, F. Wotawa, Protocol conformance testing a SIP registrar: an industrial application of formal methods, in: M. Hinchey, T. Margaria (Eds.), Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods, IEEE, London, UK, 2007, pp. 215–224.
[35] L. Heerink, J. Feenstra, J. Tretmans, Formal test automation: the conference protocol with PHACT, in: H. Ural, R.L. Probert, G. von Bochmann (Eds.), Proceedings of the 13th International Conference on Testing Communicating Systems, IFIP Conference Proceedings, vol. 176, Kluwer, 2000, pp. 211–220.
[36] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, M. Veanes, Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, Tech. Rep. MSR-TR-2005-59, Microsoft Research, 2005.
[37] M. Botinčan, V. Novaković, Model-based testing of the conference protocol with spec explorer, in: Proceedings of the Ninth International Conference on Telecommunications, IEEE, 2007, pp. 131–138.
[38] M. Weiglhofer, F. Wotawa, Random vs. scenario-based vs. fault-based testing: an industrial evaluation of formal black-box testing methods, in: Proceedings of the Third International Conference on Evaluation of Novel Approaches to Software Engineering, Funchal, Madeira, Portugal, 2008, pp. 115–122.