SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–33
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Killing strategies for model-based mutation testing

Bernhard K. Aichernig[1*], Harald Brandl[2], Elisabeth Jöbstl[1],
Willibald Krenn[3], Rupert Schlick[3], Stefan Tiran[1,3]

[1]*Institute for Software Technology, Graz University of Technology, Graz, Austria*
[2]*AVL List GmbH, Hans-List-Platz 1, Graz, Austria*
[3]*Austrian Institute of Technology, Vienna, Austria*

SUMMARY

This article presents the techniques and results of a novel model-based test case generation approach that automatically derives test cases from UML state machines. The main contribution of this article is the fully automated fault-based test case generation technique together with two empirical case studies derived from industrial use cases. Also, an in-depth evaluation of different fault-based test case generation strategies on each of the case studies is given and a comparison to plain random testing is conducted. The test case generation methodology supports a wide range of UML constructs and is grounded on the formal semantics of Back's action systems and the well-known input-output conformance relation (ioco). Mutation operators are employed on the level of the specification to insert faults and generate test cases that will reveal the faults inserted. The effectiveness of this approach is shown and it is discussed how to gain a more expressive test suite by combining cheap but undirected random test case generation with the more expensive but directed mutation-based technique. Finally, an extensive and critical discussion of the lessons learnt is given as well as a future outlook on the general usefulness and practicability of mutation-based test case generation.

Received . . .

KEY WORDS: test case generation; model-based testing; mutation testing; random testing; ioco; action systems; Unified Modeling Language; UML

1. INTRODUCTION

Today's dependable computer-based infrastructure rapidly grows in complexity due to a continuous evolution towards very large, heterogeneous, highly dynamic and ubiquitous computer systems. This is a serious challenge to the task of engineering trustworthy systems: the more complex a system is, the more difficult is the verification of its correctness. It seems that despite the many advances in automated verification, i.e., in model checking and theorem proving, the demand for new features and flexibility always creates systems that establish the next limits for automated verification. However, the situation is not hopeless.

Where formal verification is not feasible, a formal testing approach may be applied. Such formal testing comprises formal models of the system under test and a testing theory that captures the essential properties of the testing technique. The latter includes, most notably, the semantics of

*Correspondence to: Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/2, 8010 Graz, Austria. E-mail: aichernig@ist.tugraz.at

Contract/grant sponsor: EU FP7 project Model-based Generation of Tests for Dependable Embedded Systems (MOGENTES); contract/grant number: ICT-216679

the communication between the tester and the system under test (SUT), and a precise definition of conformance between the SUT and a model. Hence, formal testing is model-based testing with solid theoretical foundations. It was first advocated by Gaudel [1, 2].

The approach presented here is based on formal mutation testing [3, 4]. Instead of programs, however, abstract models of the SUT are mutated. All mutations are introduced by so called mutation operators that alter the models syntactically. Mutated models represent faulty designs and the goal of the presented approach is to generate test cases that can kill these faulty designs. Put differently, a sequence of actions is sought that triggers different behaviour between the original and the mutated model. When executed on the SUT, the tests will detect if a mutated model has been implemented.

The advantage of this style of testing is that it is fault-centred. In contrast to covering certain states or transitions in a model, the focus is on covering possible flaws in the design of a system. This testing strategy is very fine grained: whatever can be considered a fault on the modelling level can be covered by a test case.

However, there also is a price to pay. Model-based mutation testing is computationally expensive. Hundreds of mutants need to be analysed and the test case generation process involves an equivalence check of two models that is undecidable in general. The situation is even more complex for non-deterministic models. Simple equivalence checking is not sufficient in this case, instead a conformance relation that is a preorder needs to be checked. Can this work in practice? State of the art in industry answers negatively. As of today there are no commercial model-based mutation testing tools available.

In the EU FP7 project MOGENTES[†] this challenge has been accepted and a model-based mutation testing tool for UML state transition diagrams has been developed. To ease the task of test data generation and to stay formal, the tool first maps UML to labeled transition systems (LTS). Besides defining a formal meaning for semantic variation points in the UML standard the mapping also provides access to existing testing theories on LTS. In particular Tretmans' testing theory based on the conformance relation *ioco* [5] has been selected as it supports partial and non-deterministic models. This automated mapping process is described in detail elsewhere [6, 7].

The test case generator essentially performs an *ioco* check. The tool takes an original and a mutated model as input and explores their state space in order to detect a difference in behaviour. The exploration depth is limited by a given boundary. If the mutated model produces an output that is not allowed by the original model, non-conformance is detected. In this case a test case is generated that shows the difference in behaviour. If executed on the SUT, the test checks whether the faulty behaviour has been implemented. The general idea and its extension to hybrid system testing has been presented previously [8]. The conformance checker itself has also been discussed in detail in previous work of the authors [9].

Focus of this article are different test case generation strategies once non-conformance has been detected. The strategies vary in the number of generated test cases, the generation time, and in the structure of the test cases built. The latter refers to the fact that some strategies generate linear test cases, others adaptive ones. A variety of test case selection strategies has been implemented and their respective effectiveness has been compared with the help of two case studies. The first case study is a car alarm system provided by the industrial partner Ford. As a second, larger case study, the electronic control unit of a wheel-loader bucket arm has been used.

The aim has been to explore the space of possible test case generation strategies and to investigate, on non-trivial models, the cost and bug-detection capabilities associated to each of the strategies. The first, most straightforward and expensive, strategy presented here is to cover all paths to the point of non-conformance in the LTS. This strategy naturally evolves to one for generating adaptive test cases. Next, the ability to check whether existing test cases can already kill a mutant, is added. Finally, a strategy where a set of random tests is generated in a first step and then improved by mutation-based test-case generation in a second step is proposed. This results in a total of six different mutation testing strategies.

[†] `http://www.mogentes.eu/`

DOI: 10.1002/stvr

In order to compare these strategies to more traditional ways of test case generation, pure random testing and test case generation with hand-written test purposes have been added to the experiments. The latter is implemented via an interface to the CADP[‡] tools. It provides access to the TGV test case generator [10] as well as to model checkers and simplifiers of CADP.

The main questions to be investigated in this article are:

- *What is the best strategy to select test cases?* The strategies will be compared with respect to the number of generated test cases, as well as to their ability to find bugs.
- *Is mutation testing better than random testing?* The aim is to find as many bugs as possible with a relatively small test suite. The comparison to random testing is interesting because the fault-based technique requires sophisticated algorithms and it is necessary to justify that the investment pays off. The relation to other structural coverage criteria, like e.g. transition coverage, is out-of-scope of this study.
- *How efficient is a test case generation strategy?* Details on the run-times of each test case generation strategy are reported.
- *How severe is the equivalent mutants problem?* Equivalent mutants show the same behaviour as the original. Hence, no test case can possibly kill them and trying to generate one will fail. The effects on the efficiency of the algorithms is investigated.
- *Do partial models help?* In the larger case study of the wheel-loader, two partial models are used. Each model is focusing on different functional aspects of the system. Partial models result in a smaller state space but might compromise bug detection capabilities, hence the effects of partial models are explored.
- *Does the combination of random testing and mutation testing help?* Random testing and mutation testing are orthogonal in the sense that the first is cheap and undirected while the latter is expensive and directed. This work investigates the question whether a combination of both increases the fault detection capabilities. The conjecture is that a few long random tests will quickly detect most of the trivial bugs and the fault-based strategies will cover the more subtle faults.
- *Given a set of faulty implementations, can all known bugs be revealed?* Both case studies have been implemented in Java and a set of mutated implementations has been created and cleaned of equivalent mutants. The percentage of the remaining non-equivalent mutants that are killed by the generated test suites (mutation scores) is reported.

This article is an extended version of a previous conference paper [11]. The main new contribution is the large wheel-loader case study that challenges the mutation-based test case generation approach. To give the reader an idea of the complexity involved, it shall be mentioned that a full state machine model of the wheel loader consists of six orthogonal (parallel) regions, multiple nested states, and timing constraints. In addition a significantly extended presentation of the test case generation strategies is given, which includes further examples.

The rest of the article is structured as follows. Section 2 presents a first case study and sketches the mapping from UML to LTS. Section 3 introduces a formal characterisation of the resulting test cases and an algorithm for the generation of adaptive test cases. Section 4 explains the different test selection strategies and Section 5 presents the empirical results gained when applying them to the two case studies. Before concluding the article in Section 8, the experimental results are discussed in Section 6 and related research is reviewed in Section 7.

2. A UML MODEL

In order to demonstrate the basic concepts of the mutation-based test case generation approach, a simplified version of a car alarm system (CAS) is used. The example is taken from Ford's automotive

[‡] `http://www.inrialpes.fr/vasy/cadp/`

DOI: 10.1002/stvr

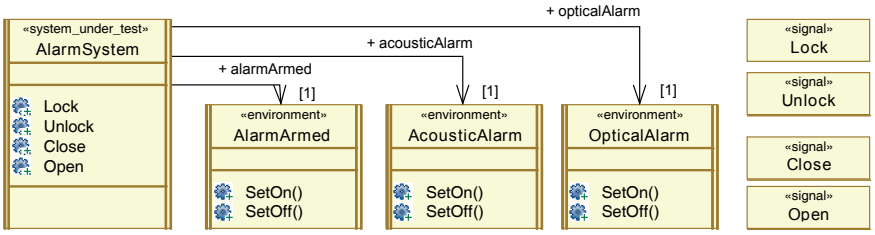

Figure 1. Testing interface of the car alarm system.

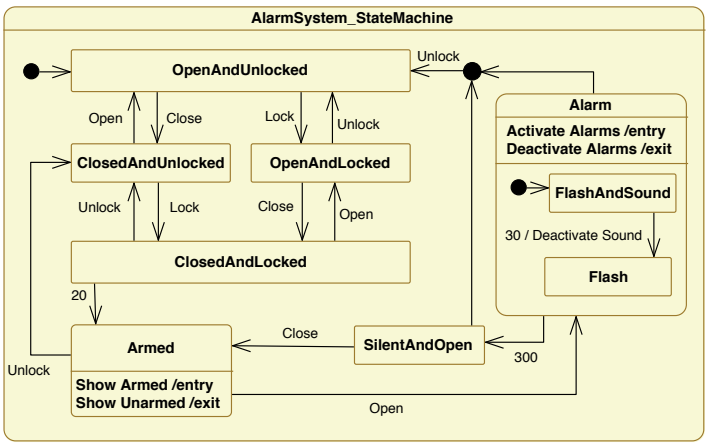

Figure 2. State machine of the car alarm system.

demonstrator within the MOGENTES project. The following requirements serve as the basis for the UML test model:

- **R1** **Arming** The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment, and all doors are closed.
- **R2** **Alarm** The alarm sounds for 30 seconds if an unauthorised person opens the door, the luggage compartment, or the bonnet. The hazard flasher lights will flash for five minutes.
- **R3** **Deactivation** The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

2.1. Testing Interface

Figure 1 shows the class diagram of the car alarm system's UML model. According to the requirements the diagram comprises four classes and four signals. The class *AlarmSystem* is labelled as SUT and may receive any of the *Lock*, *Unlock*, *Close*, or *Open* signals. At the same time, the SUT calls methods of the classes *AlarmArmed*, *AcousticAlarm*, and *OpticalAlarm* – all of them labelled as being part of the environment. In this way, the diagram of Figure 1 specifies the possible observations (all calls to methods being part of the environment) and the stimuli the SUT can take (all signals). Hence, the diagram specifies the testing interface.

2.2. State Machine

Figure 2 shows the CAS state machine diagram. From the state *OpenAndUnlocked* one can traverse to *ClosedAndLocked* by closing all doors and locking the car. Actions of closing, opening, locking, and unlocking are modelled by corresponding signals *Close*, *Open*, *Lock*, and *Unlock*. As specified in requirement R1, the alarm system is armed after 20 seconds in *ClosedAndLocked*. Upon entry of the *Armed* state, the model calls the method *AlarmArmed.SetOn*. Upon leaving the state, which

DOI: 10.1002/stvr

Table I. Number of mutated models per mutation operator.

Mutation Operator	# CAS	Upper Bound	
Set Guard To False	19	$O(t)$	t...# of transitions
Remove Entry Action	3	$O(s_{en})$	s_{en} ... # of states with entry actions
Remove Exit Action	3	$O(s_{ex})$	s_{ex} ... # of states with exit actions
Remove Signal Trigger	12	$O(t_{st})$	t_{st} ... # of signal triggered transitions
Remove Time Trigger	3	$O(t_{tt})$	t_{tt} ... # of time triggered transitions
Mutate Transition Signal Events	36	$O(t_{st} \cdot s)$	t_{st} ... # of signal triggered transitions, s ... # of signals

can be done by either unlocking the car or opening a door, *AlarmArmed.SetOff* is called. Similarly, when entering the *Alarm* state, the optical and acoustic alarms are enabled. When leaving the alarm state, either via a timeout or via unlocking the car, both acoustic and optical alarm are turned off. When leaving the alarm state after a timeout (cf. requirement R2) the chosen treatment of the underspecification in the requirements is that the system returns to an armed state only in case it receives a close signal. Turning off the acoustic alarm after 30 seconds, as specified in requirement R2, is reflected in the time-triggered transition leading to the *Flash* sub-state of the *Alarm* state.

2.3. Mutations

Since test cases that cover particular fault models are desired, the specification is mutated and test cases are created that can identify the different behaviours. Applying the six mutation operators listed in Table I to the CAS model of Figure 2 yields a total of 76 mutated UML state machines. Table I also gives an upper bound on how often each mutation operator can be applied.

The first mutation operator sets the guard of a transition to false[§]. This removes the behaviour of the transition. Sometimes, however, behaviour is not linked to a specific transition but to entry and exit actions. These behaviours occur whenever a certain state is entered/left, regardless of the transition used. The mutation operators that remove such behaviours are called *Remove Entry Action* and *Remove Exit Action*. Similarly, for each signal/timer triggering a transition there is a mutation operator removing it (*Remove Signal Trigger* and *Remove Time Trigger*). Finally, there is a mutation operator that replaces the signal triggering a transition by an arbitrary other (but compatible) signal. Due to combinatorial effects this operator is responsible for nearly half of the generated mutants.

Mutation testing relies on two assumptions: (1) competent engineers write almost correct code and (2) there exists a coupling effect so that complex errors will be found by test cases that can detect simple errors. This is the main reason that only first-order mutants are used for test-case generation, i.e. each mutated state machine results from applying a single mutation operator to a single location in the model. The number of first-order mutants grows in the same order as the number of mutatable elements (model size) grows, unless the mutation operator is parametrised with other model elements. In the latter case the growth increases from linear to quadratic given one parameter.

Note that the approach presented in this article also addresses the well-known problem of equivalent mutants. In model-based mutation testing, a conformance check between the original and the mutant is conducted. If non-conformance between the mutant and the original is found, a test case is generated that kills the mutant. If it is impossible to discover any non-conforming behaviour up to a certain search depth, the mutant is considered to be equivalent up to that depth and no test case is generated.

2.4. From UML to LTS via Action Systems

Since the algorithms for test case generation work on the level of labeled transition systems (LTS, [12]), the UML state machine model defined by the user has to be converted into an LTS. This

[§]This may lead to a transition transforming into a self-loop as the model will 'swallow' the trigger event.

DOI: 10.1002/stvr

process is fully automated and the tool accepts a wide range of input models. Supported UML features include nested states, orthogonal regions, (final-, initial-, junction-, and choice pseudo) states, entry and exit actions, transitions with effects, triggers with change/signal/call events, as well as a large set of OCL constraints as transition guards. Supported OCL constructs are standard logic operators, select, collect, exists, forall, oclIsInState, and literals (integer, Boolean). Besides enums, signals, Booleans, integers, there is also support of active or passive classes with methods, member fields, signal reception, associations, and a restricted form of inheritance. The conversion process itself comprises the following steps:

1. The given UML model is transformed into a labeled and object-oriented intermediate representation called Object-Oriented Action System (OOAS). It is based on Back's action system [13] formalism. During this step the exact UML semantics are set, e.g., the treatment of UML events and orthogonal regions. As a result, an executable model of the system is derived.
2. The object-oriented intermediate representation is further simplified into a non-object-oriented (but still labeled) version of the action system.
3. The action system derived in the second step is explored. This process yields the LTS of the UML model: Roughly speaking the execution of one labeled action within the action system adds one transition to the LTS.

Further information about this conversion can be found in previous work [6, 7] of the authors.

Applying the tool to the car alarm system (see Figure 2) yields the LTS shown in Figure 3. All labels in the system are prefixed by either *obs* (observable) or *ctr* (controllable) with observables being system outputs and controllables being inputs of the car alarm system. As explained in Section 2.1, these prefixes are automatically deduced from the diagram specifying the testing interface (see Figure 1). An exception to this is the label *obs after* that stands for an observable timeout. Briefly speaking, *after* is observable because it represents a special case of the *ioco* timeout δ that is defined observable within the *ioco* theory (see Section 3.1).

3. TEST CASE GENERATION

In this section, the test case generation technique is presented. First, the conformance relation that enables the comparison of original and mutated specifications is explained. Then, the actual implementation of the conformance checker is outlined. This includes a discussion of the properties of all generated test cases and the algorithm for test case selection.

3.1. Input-Output Conformance

Conformance relations are needed to determine if a SUT behaves correctly regarding a given specification. In order to decide conformance, some testing hypotheses have to be stated [1]. One is that the implementation can be represented with the same formalism as the specification. In the presented tool this is always the case since the conformance between two specifications (an original and a mutated version) is decided. The mutated testing model serves as SUT, while the non-mutated testing model serves as specification. In order to be able to deal with partial system models, the *ioco* relation [5] is used.

The trace semantics of an action system yields a Labeled Transition System (LTS) M, see Section 2.4 and previous work [9]. A labeled transition system is defined as tuple $\langle S, L, T, s_0 \rangle$ where

- S is a countable set of states
- $L = L_U \cup L_I$ is a countable set of labels divided into input labels L_I and output labels L_U such that $L_I \cap L_U = \emptyset$
- $T \subset S \times (L \cup \{\tau\}) \times S$ is the transition relation, and
- $s_0 \in S$ is the initial state.

DOI: 10.1002/stvr

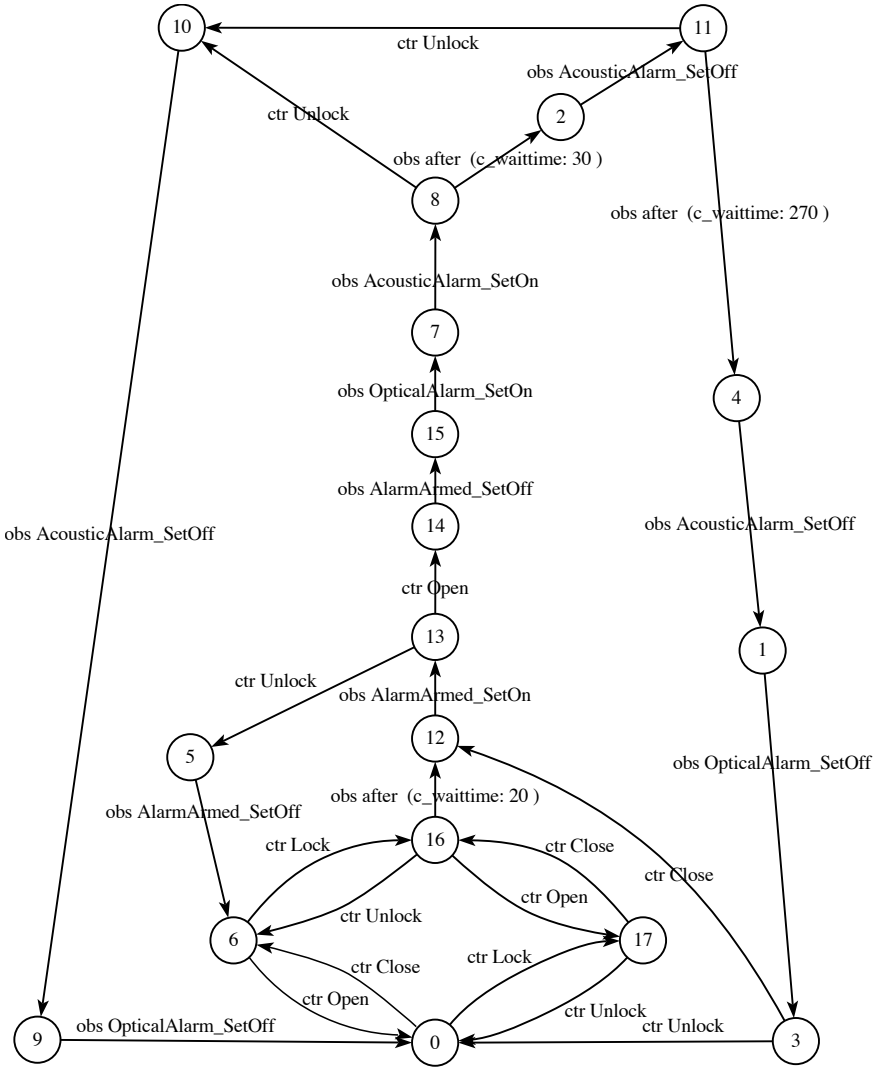

Figure 3. Labeled transition system of the car alarm system.

The special label τ denotes an internal action. For a given LTS the following notation is used, where $s, s', s_i \in S, S' \subseteq S, a_i \in L, \sigma \in L^*$.

$$s \xrightarrow{a} s' \;=_{df}\; (s, a, s') \in T \tag{1}$$

$$s \xrightarrow{a} \;=_{df}\; \exists s' \bullet (s, a, s') \in T \tag{2}$$

$$s \not\xrightarrow{a} \;=_{df}\; \nexists s' \bullet (s, a, s') \in T \tag{3}$$

$$s \overset{\epsilon}{\Rightarrow} s' \;=_{df}\; s = s' \vee \exists s_0 \ldots s_n \bullet s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_{n-1} \xrightarrow{\tau} s_n = s' \tag{4}$$

$$s \overset{a}{\Rightarrow} s' \;=_{df}\; \exists s_1, s_2 \bullet s \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Rightarrow} s' \tag{5}$$

$$s \overset{\sigma}{\Rightarrow} s' \;=_{df}\; \exists s_0, \ldots, s_n \bullet s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} s_n = s' \textit{with } \sigma = a_1 \ldots a_n \tag{6}$$

$$s \overset{\sigma}{\Rightarrow} \;=_{df}\; \exists s' \bullet s \overset{\sigma}{\Rightarrow} s' \tag{7}$$

$$s \;after\; \sigma \;=_{df}\; \{s' \mid s \overset{\sigma}{\Rightarrow} s'\} \tag{8}$$

DOI: 10.1002/stvr

Equation 1 introduces $s \xrightarrow{a} s'$ as an alternative notation for the transition (s, a, s'). Then, $s \xrightarrow{a}$ states that there starts a transition labeled with a from state s leading to some arbitrary successor state (Equation 2). Equation 3 states the contrary, i.e., that there does not exist such a transition. $s \stackrel{\epsilon}{\Rightarrow} s'$ means that either both states are the same ($s = s'$) or that there exists a set of intermediate states such that s' is reachable from s via a sequence of internal (τ) transitions only (Equation 4). $s \stackrel{a}{\Rightarrow} s'$ means that it is possible to reach state s' from state s by action a - with internal transitions before or after a as required (Equation 5). Equation 6 states that state s' is reachable from state s by a trace σ (a sequence of transitions labeled with $a_1 ... a_n$). Internal actions may be required before, after or between the transitions of this trace. Equation 7 defines that there exists a state s' that is reachable from s by trace σ. The relation *after*¶ (Equation 8) determines the set of states reachable after trace σ starting from state s. Moreover, an LTS M has finite behaviour if all traces have finite length. It is deterministic if $\forall \sigma \in L^* \bullet |s_0 \textbf{ after } \sigma| \leq 1$ holds.

For the *ioco* relation, SUTs are considered to be weakly input-enabled, i.e., all inputs (possibly preceded by τ transitions) are enabled in all states: $\forall a \in L_I, \forall s \in S \bullet s \stackrel{a}{\Rightarrow}$. This class of LTS is referred to as $IOTS(L_I, L_U)$ where $IOTS(L_I, L_U) \subset LTS(L_I \cup L_U)$. A state s from which the system cannot proceed without additional inputs from the environment is called *quiescent*, denoted as $\delta(s)$. In such a state, all output and internal events are disabled: $\forall a \in L_U \cup \{\tau\} \bullet s \stackrel{a}{\not\rightarrow}$. For observing quiescence, the transition relation T is extended by adding self-loops with the special label δ at quiescent states: $T_\delta =_{df} T \cup \{(s, \delta, s) \mid s \in S \wedge \delta(s)\}$. Let M_δ be the LTS over the alphabet $L \cup \{\tau, \delta\}$ resulting from adding δ self-loops to an LTS M. The deterministic automaton obtained via subset construction from M_δ is called *suspension automaton* $\Gamma = \langle S_\Gamma, L \cup \{\delta\}, T_\Gamma, s_{0_\Gamma} \rangle$ [5].

The behaviour of Γ is defined via the set of suspension traces:

$$Straces(M_\delta) =_{df} \{\sigma \in (L \cup \{\delta\})^* \mid s_{0_\Gamma} \stackrel{\sigma}{\Rightarrow}\}$$

The set of outputs that can occur in a state $s \in S$ (in a set of states $S' \subseteq S$ respectively) is defined as follows:

$$out(s) =_{df} \{a \in L_U \mid s \xrightarrow{a}\} \cup \{\delta \mid \delta(s)\}$$
$$out(S') =_{df} \bigcup_{s \in S'} out(s)$$

The *ioco* relation states that for all suspension traces in the specification, the outputs of the implementation after such a trace must be included in the set of outputs produced by the specification after the same trace. Formally, for an implementation model $i \in IOTS(L_I, L_U)$ and a specification $s \in LTS(L_I \cup L_U)$ the relation *ioco* is defined as follows:

$$i \textit{ ioco } s =_{df} \forall \sigma \in \textit{Straces}(s) \bullet \textit{out}(i \textit{ after } \sigma) \subseteq \textit{out}(s \textit{ after } \sigma) \tag{9}$$

3.2. Conformance Checking

A tool for checking the input-output conformance of two system models named *Ulysses* has been implemented in SICStus Prolog‖. Figure 4 depicts the computation steps of this tool. *Ulysses* expects two labeled action systems as input: (1) a system specification AS and (2) a mutated version of the same specification AS^M. In Section 2.4 the transformation from a UML model to an LTS has already been presented. The last step of this conversion is actually performed in *Ulysses* by exploring a given labeled action system. Previous work [9] can be consulted for further details. The LTS is enriched by quiescence and subsequently converted into a deterministic automaton. Executing these steps, which are depicted in the first box of Figure 4, result in the so-called *suspension automaton* of the model.

¶Not to be confused with the timeout observation *after (t)* in Figure 3.
‖`https://sicstus.sics.se/`

DOI: 10.1002/stvr

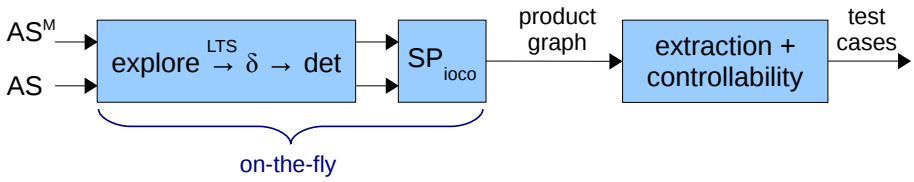

Figure 4. The computation steps of *Ulysses*.

Ulysses generates the suspension automata for both input models AS and AS^M. Afterwards, the *ioco* check for these two models is performed (see the central box in Figure 4). The checker generates a product graph from which controllable test cases (last box in the *Ulysses* process) are extracted. Note that the calculation of the suspension automata and the synchronous product calculation modulo *ioco* (SP_{ioco}) are performed on-the-fly, which means that the automata are only unfolded as required by the conformance check [9].

Example 1. Consider the example of Figure 5. The first two LTSs show the partial suspension automata of the CAS specification and of one of its mutants. The partial specification defines the disarming of the alarm system when the car is unlocked. In the mutant, the disarming via an *Unlock* is not possible as the controllable action *Unlock* has been replaced by *Open*. After making the mutant input-complete the *Unlock* action is added as a self-loop in the particular state. For readability of Figure 5, the non-input-enabled version of the mutant is shown.

The mutant is not *ioco* to the specification because the subset inclusion of the observations after the trace

$$tr = \langle Close, Lock, after(20), AlarmArmed\_SetOn, Unlock \rangle$$

does not hold, i.e.,

$$out(Mutant\ after\ tr) = \{\delta\} \not\subseteq \{AlarmArmed\_SetOff\} = out(Specification\ after\ tr)$$

The product LTS in Figure 5 is computed as follows. Like in conventional synchronous product calculation, both transition systems proceed to their next state on common events. Additionally, if the mutant provides an input which is not available in the specification then the product terminates after such an event in a pass state. In the example, *Open* is such an unspecified controllable event. Conversely, when an input is available only in the specification then, due to the input-enabledness of the mutant, the mutant stays in the current state and the specification moves to the next state. After output events that occur in the specification but not in the mutant, the product terminates in a pass state. Finally, after an unspecified output of the mutant the product terminates in a fail state.

In the product LTS pass and fail states are denoted by accordingly labeled self-transitions. If a product LTS contains a fail state, it is possible to derive a controllable test case that is able to detect this faulty behaviour. Note that the selected test cases are positive (implicit fail verdicts) and that they cover the faulty behaviour of a mutant in order to prove the absence of the mutation. □

3.3. Test Case Selection and Coverage

In the following, the required properties of the selected test cases are stated and then a selection algorithm for adaptive test cases is presented.

Given a product LTS (S^P, L^P, T^P, s_0^P) being the result of an *ioco* check and a set $Fail \subset S^P$ denoting the set of fail states. An unsafe state is defined as a state in which an unspecified output can occur.

$$Unsafe =_{df} \{s \in S^P \mid s \xrightarrow{a} f \wedge f \in Fail \wedge a \in L_U\}$$

Unsafe states play the central role in any test case generation strategy presented here, since they represent the test goals that should be covered by the set of generated test cases. In the following the general properties of a generated test case $TC = (S^{TC}, L^{TC}, T^{TC}, s_0^{TC})$, selected from a product LTS, are presented.

DOI: 10.1002/stvr

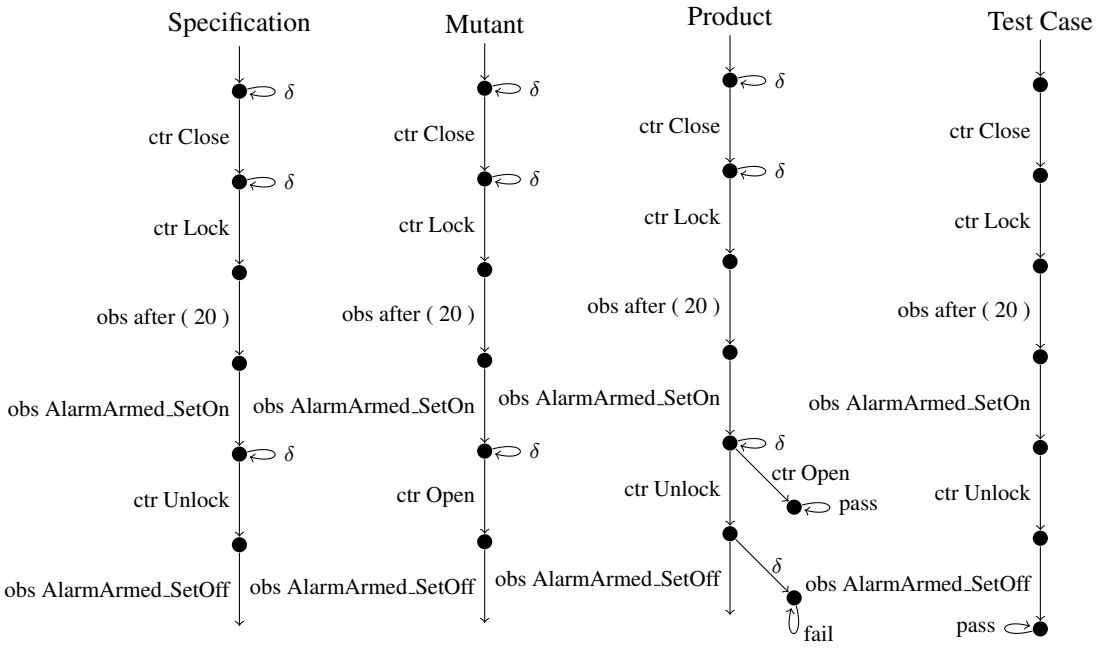

Figure 5. The two LTSs on the left show parts of the suspension automata of the CAS specification and of one of its mutants. The two LTSs on the right depict the resulting *ioco* product and a selected test case.

$$S^{TC} \setminus \{pass, inconc\} \subseteq S^P \wedge L^{TC} \subseteq L^P \wedge s_0^{TC} = s_0^P \quad \text{(inclusion)}$$

$$s \overset{a}{\not\rightarrow} \iff s \in \{pass, inconc\} \quad \text{(sink states)}$$

$$s = pass \iff \exists u \in Unsafe \bullet u \overset{a}{\rightarrow} s \wedge s \notin Fail \ \wedge a \in L_U \quad \text{(passing)}$$

$$(s \in S^{TC} \wedge s \overset{a}{\Rightarrow} \wedge s \overset{b}{\Rightarrow}) \implies (a, b \in L_I \wedge a = b) \vee a, b \in L_U \quad \text{(controllability)}$$

$$\exists!\ u \in S^{TC} \bullet (u \in Unsafe) \wedge u \overset{a}{\rightarrow} pass \quad \text{(test goal unique)}$$

$$\exists\ \sigma \bullet s_0^{TC} \overset{\sigma}{\Rightarrow} u \wedge u \in Unsafe \quad \text{(test goal reachable)}$$

A test case is a sub-transition system of the calculated product extended with two additional verdict states *pass* and *inconc* (inclusion). In the test case, a sink state is always a verdict state *pass* or *inconc*. Note, fail verdicts are implicit and not included (sink states). The pass verdict is characterised by successfully passing an unsafe state (passing). A test case does not contain choices over controllables (controllability) [5]. Reaching an unsafe state is the test goal of mutation testing. Hence, exactly one unsafe state precedes a pass verdict state (test goal unique) per test case. Finally and most importantly, the test case must be able to reach its test goal, i.e., the unsafe state (test goal reachable).

There are two kinds of test cases that may reach the test goal, i.e., a given unsafe state: First, a linear test case that includes one path to the unsafe state. Second, a branching adaptive test case that may include several paths to an unsafe state. In the following, the properties of the two kinds are presented.

DOI: 10.1002/stvr

Algorithm 1 $getTC(s, Goal) : S \times \mathcal{P}(S) \mapsto \mathcal{P}(S \times (L \cup \{\delta, inconc\}) \times S)$

1: **if** $s \in Goal$ **then**
2: **return** $\emptyset$
3: **else**
4: $\sigma := shortestTrace(s, Goal)$
5: **if** $\sigma = \langle\rangle$ **then**
6: **return** $\{(s, inconc, s)\}$
7: **else**
8: $E := edges(\sigma)$
9: $Goal := Goal \cup states(E)$
10: **return** $E \cup \bigcup\{\{(s_1, a, s_2) \in T^P\} \cup getTC(s_2, Goal) \mid$
$\exists a', s_2' \bullet (s_1, a', s_2') \in E \wedge a, a' \in L_U \wedge a \neq a' \wedge s_2 \notin Fail\}$
11: **end if**
12: **end if**

3.3.1. Linear Test Cases are necessary if the target test harness does not support branching behaviour. The following two additional properties characterise linear test cases:

$$\exists!\ \sigma \bullet s_0^{TC} \stackrel{\sigma}{\Rightarrow} u \wedge u \in \mathit{Unsafe} \qquad \text{(linear test case)}$$

$$s^{TC} \stackrel{a}{\rightarrow} \mathit{inconc} \iff s^{TC} \stackrel{\sigma}{\Rightarrow} u \wedge u \in \mathit{Unsafe} \ \wedge \exists s' \bullet ((s^{TC}, a, s') \in T^P \wedge a \in L_U \ \wedge a \notin \mathit{hd}(\sigma)) \qquad \text{(inconclusive linear)}$$

Such a test case contains exactly one path to the unsafe state (linear test case). Since a model's behaviour may branch, an observation may lead away from the linear path. In this case, the test has to be stopped with an inconclusive verdict (inconclusive linear). Here, the head operator *hd* returns the first element in the trace σ.

3.3.2. Adaptive Test Cases integrate several paths to the unsafe state into one test case. They only give an inconclusive verdict if it is impossible to reach the unsafe state:

$$s^{TC} \stackrel{a}{\rightarrow} \mathit{inconc} \iff s^{TC} \stackrel{\sigma}{\Rightarrow} u \wedge u \in \mathit{Unsafe} \ \wedge \exists s' \bullet ((s^{TC}, a, s') \in T^P \wedge a \in L_U \ \wedge s' \stackrel{\sigma'}{\not\Rightarrow} u) \qquad \text{(inconclusive adaptive)}$$

Note the difference from the linear test case in the last conjunct. A linear test case reports inconclusive if an observation leads away from the single path to the unsafe state. In contrast, in the adaptive case, inconclusive is only reported if after an observation it is impossible to reach the unsafe state.

In the following, the extraction of such an adaptive test case from a given product LTS (see Section 3.2) is described. An adaptive test case is a selection of transitions of the product LTS augmented with the verdicts *pass* and *inconc* (inconclusive). Only positive test cases are generated, i.e., fail verdicts are implicit and not included.

Algorithm 1 describes the procedure $getTC$. It is the recursive part of the test case extraction and selects transitions from the product LTS. In the following description, set variables are denoted by capital letters. As an input, Algorithm 1 requires a state s of the product LTS and and a set of goal states $Goal$. Initially, the algorithm is called with the initial state s_0 and the unsafe state u that shall be covered by the resulting test case: $getTC(s_0, \{u\})$. The algorithm searches for the shortest trace from the initial state to an *unsafe* state and recursively extends this trace for all branching output transitions. More precisely: at first the current state is checked, whether it is already a goal state (Line 1). If this is the case, the empty set is returned. Otherwise, a trace leading to a goal state is searched. In Line 4 the shortest trace between the given start state s and one state in the set of goal states is determined. Note that there can exist several shortest traces of the same length. In this case, one of them is chosen non-deterministically. If there exists no such trace then

DOI: 10.1002/stvr

all observations in state s terminate in an inconclusive state. Otherwise the transitions E along trace σ (Line 8) are part of the test case. The transitions of a trace are determined by the function: $edges : (L \cup \delta)^* \mapsto \mathcal{P}(S \times (L \cup \delta) \times S)$. In Line 9, the set of states in E obtained by the function $states$ are added to the set of goal states. Hence, the goal becomes to reach one state of the test case which in turn leads to an unsafe state. Next, the test case is recursively extended by all branching observable events along the trace σ (Line 10). Here, the set comprehension creates a set of sets of transitions. Each set corresponds to a subgraph in the test case after a branching observation. The recursive algorithm terminates when all branching outputs have been processed.

The return value of Algorithm 1 is of type $\mathcal{P}(S \times (L \cup \{\delta, inconc\}) \times S)$. It already incorporates *inconc* verdicts represented as self-looped transitions. In order to accomplish a valid, adaptive test case TC_u, *pass* verdicts still have to be added. This is accomplished by inserting transitions to pass states as follows:

$$TC_u =_{df} getTC(s_0, \{u\}) \cup \{(u, a, pass) \mid \exists s \bullet u \xrightarrow{a} s \ \wedge a \in L_U \ \wedge s \notin Fail\} \quad \text{(adaptive test case)}$$

Hence, an adaptive test case is of type $\mathcal{P}(S \times (L \cup \{\delta, inconc\}) \times (S \cup pass))$.

4. MUTATION KILLING STRATEGIES

The preceding section provided the theoretical basis for the presented model-based mutation testing framework. This section describes eight different strategies called S1 to S8 to select test cases. Strategies S1 to S6 are mutation-based, while S7 and S8 serve as benchmarks for the mutation-based strategies. Strategy S7 is better known as random testing and strategy S8 uses manually designed test cases.

4.1. S1 - All Paths Killing Strategy

Strategy S1 first converts the product graph obtained during the *ioco* check to a tree. The unfolding of the graph is implemented via a breadth-first search beginning from the graph's initial state. To avoid infinite loop unfolding, the following two stopping criteria are defined: (1) The algorithm stops to explore a certain path if a user-specified maximum depth d is reached. (2) The exploration will also stop if some state of the product graph has been visited the n^{th} time in that path. The bound n can also be specified by the user. Additionally, no path of the tree contains a sequence of δ transitions (quiescence). Instead, just one δ transition is added. This keeps the tree as compact as possible and saves time during test case execution.

After this unfolding, linear test cases are extracted for all paths that lead to an unsafe state in the product tree. First, the path leading from the respective unsafe state to the root node of the product tree is added to the test case via a backward traversal, cf. Section 3.3.1 (linear test case). Second, the selected path will be augmented with verdicts *inconclusive* and *pass* according to Section 3.3.1 (inconclusive linear) and (passing).

Example 2. Consider the LTSs depicted in Figure 6. The *Specification* describes a vending machine for hot beverages that delivers coffee or hot chocolate once it is ready and a coin has been inserted. Otherwise it reports *NotReady* and terminates. The *Mutant* behaves in the same way, but it delivers tea instead of hot chocolate. The *Product Graph* depicts the result of the *ioco* check between these two LTSs. Since the observable *Tea* is not specified, it leads to a fail state (depicted in dark grey). The predecessor of the fail state is the unsafe state (depicted in light grey). The graph is then unfolded to the *Product Tree* (see Figure 7). In this example, the graph has been completely unfolded except for the δ loop, which is executed at most once. From the tree, S1 then selects the two possible test cases leading to the unsafe state. *Test Case 1* represents the choice of immediately inserting a coin. *Test Case 2* does not immediately insert a coin but waits (δ) beforehand. As already explained in Section 3.3.1 (inconclusive linear), inconclusive verdicts have been added. Hence, the test cases are not only a straight path but have short branches marking behaviour leading away from the test goal of passing the unsafe state. □

DOI: 10.1002/stvr

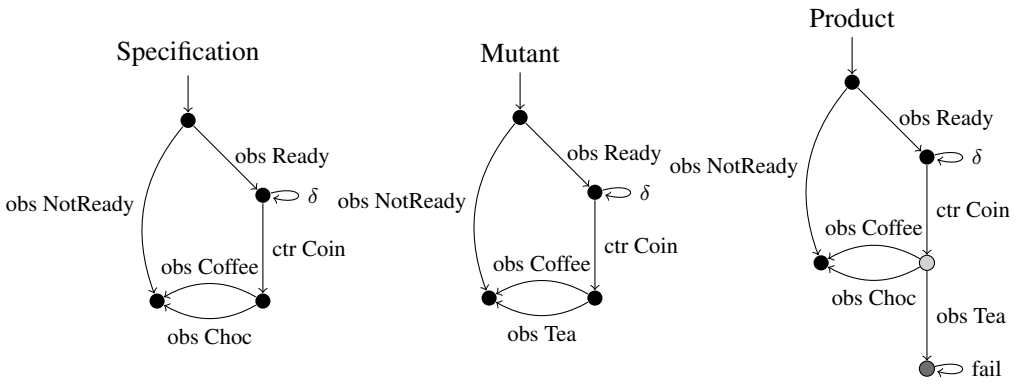

Figure 6. This example describes a vending machine for hot beverages delivering coffee or hot chocolate in the specification, but coffee or tea in the mutated version. The LTSs show the specification, the mutant, and the product modulo *ioco*.

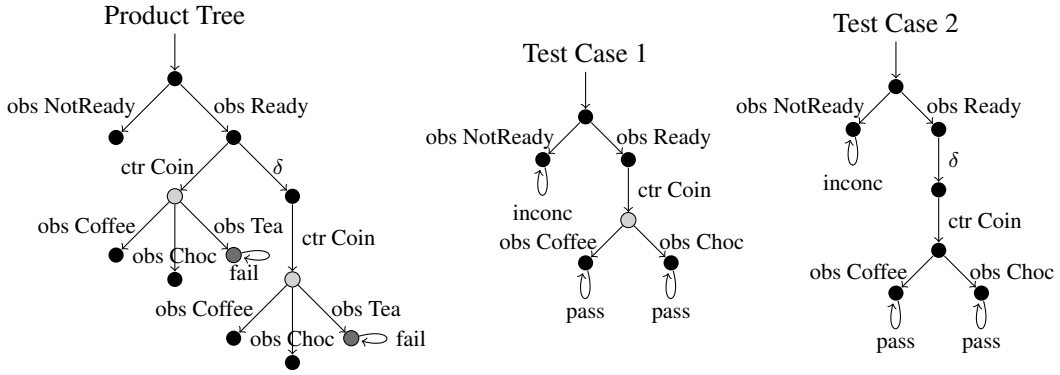

Figure 7. The first LTS depicts the tree version of the product graph shown in Figure 6 according to the unfolding procedure of killing strategy S1. The other two LTSs show the two possible test cases that were extracted from the tree by S1.

4.2. S2 - Target-Oriented Killing Strategy

The second test case generation strategy directly works on the product graph and extracts only one arbitrary linear test per unsafe state (not one for each path leading to an unsafe state as in S1). Notice that one mutation at the UML level may lead to more than one unsafe state in the LTS product graph. The test case is obtained by applying a depth-first-search from the unsafe state back to the initial state. The search depth is bounded and the first path reaching the initial state builds the test case. Finally the test case is completed with the observable events leading to pass states. It has to be noted that this approach is only suited to generate test cases for systems with deterministic output behaviour, i.e., systems that respond with exactly one output event to an input event.

Example 3. Let us consider the product LTS in Figure 6 and a bounded search depth of two. Then a depth-first-search starting from the unsafe state may yield $\langle Coin, \delta \rangle$. Since the initial state has not been reached within the search depth limit the search backtracks to the next path, i.e., $\langle Coin, Ready \rangle$. The reversal of this path in conjunction with the two observations *Coffee* and *Choc* at the end complete the test case. □

4.3. S3 - Adaptive Killing Strategy

This strategy generates a test case for every unsafe state of a mutant. It is similar to the second one (S2), except that the depth is theoretically unbounded since the test case generator applies Algorithm 1. Generated test cases are adaptive, i.e., their branching structure allows the tester to

DOI: 10.1002/stvr

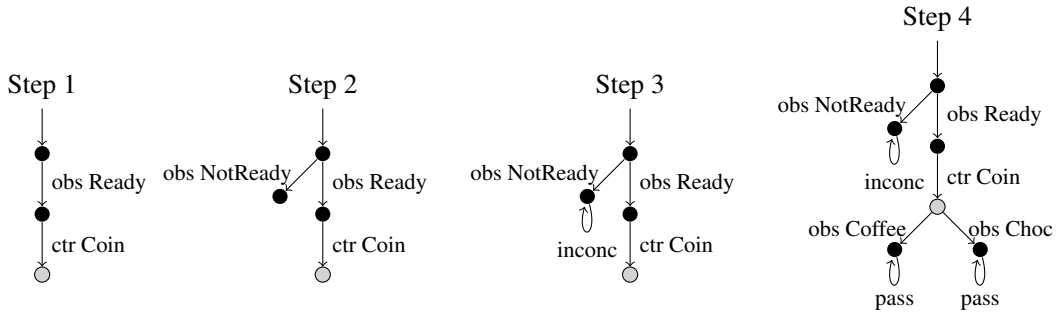

Figure 8. This figure shows the stepwise derivation of a test case from the product LTS given in Figure 6 according to S3.

follow different output responses of the implementation. The following example illustrates the test selection algorithm.

Example 4. Again, consider the vending machine example depicted in Figure 6. The selection of a test case is shown according to Algorithm 1 in four steps, see Figure 8: starting from the initial state of the product LTS, the shortest trace to the unsafe state is determined, see Step 1. Next, each transition with an observable event in the trace is completed with all other branching observable transitions. In the given trace in Step 1 there is only one observable transition, i.e., *Ready*, and according to the product LTS in Figure 6 the transition *NotReady* branches from the same state as *Ready*. Consequently, *NotReady* is added to the test case, see Step 2. The LTS in Step 2 shows the result of the first recursion level. The recursion proceeds for each newly discovered state, reached via branching transitions. In the example the state after *NotReady* is a new state. Furthermore, the set of goal states is extended to all states discovered so far, i.e., all states of Step 2. In the next recursion there exists no trace from the new state of Step 2 to any goal state. Hence, according to Algorithm 1 an *inconc* self-loop is added to the new state, see Step 3. At this point, the recursion terminates and the function *getTC* returns the LTS of Step 3. Finally, the two observable transitions leading to a pass state are added, see Step 4. □

4.4. S4 - Lazy Killing Strategy

The fourth strategy builds on S3 but avoids creating test cases for unsafe states in a given product graph which are already covered by existing test cases. In order to check if a certain unsafe state is covered by an existing test case a test purpose tp is generated for that unsafe state. Then the existing test cases are checked by computing the synchronous product with tp. This is similar to the synchronous product calculation in the tool *TGV* [10], however *TGV* computes it between test purpose and specification. A test case tc satisfies a test purpose if the product calculation prunes no events from tc, i.e., $tc \times tp = tc$.

The generated test purpose ensures that a given test case passes through the unsafe state to a pass state. Given an unsafe state u, the set of states $L2U$ which lead to u, i.e., $L2U =_{df} \{s \in S \mid \exists \sigma \bullet s \stackrel{\sigma}{\Rightarrow} u\}$ is determined. The set of pass states associated with u are $Pass =_{df} \{s \in S \mid \exists a \in L_U \bullet (u, a, s) \in T \wedge s \notin Fail\}$. The resulting test purpose tp_u has the transition relation $T_u =_{df} \{(s_1, a, s_2) \in T \mid s_1, s_2 \in L2U \vee (s_1 = u \wedge s_2 \in Pass)\} \cup \{(s, *, s) \mid s \in Pass\}$. The remaining behaviour after reaching a state $s \in Pass$ is not of interest, denoted by the $*$ label allowing any transitions to occur thereafter.

This strategy guarantees to generate at least one test case for every unsafe state. However, note that generated test cases can still be included in other test cases. This depends on the order of the processed unsafe states. If the first processed unsafe states are deep, it is theoretically possible that no generated test case is included in another one.

Example 5. Given is a coffee vending machine which accepts some amount of coins before delivering coffee. After each coin the machine notifies the user when it is ready to accept the next

DOI: 10.1002/stvr

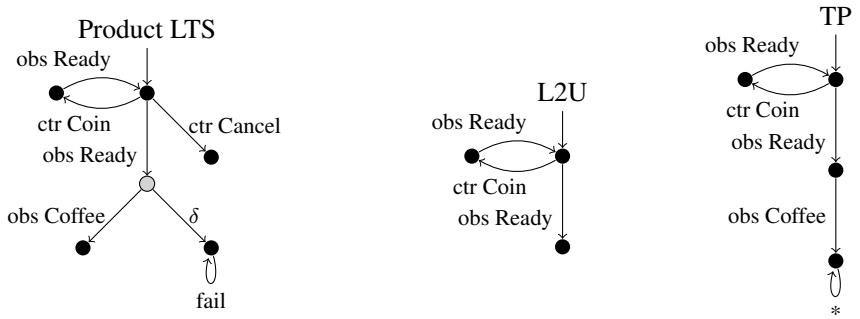

Figure 9. The first LTS shows a partial product graph of a coffee vending machine, the second LTS represents the subgraph from which every state leads to the unsafe state (grey state in the product LTS), and the third LTS shows the derived test purpose.

coin. The machine has a cancel button which aborts the coin collection process. The behaviour after pushing the cancel button is omitted. After receiving the right amount of coins and before delivering coffee the machine sends a second *Ready* signal. A mutated version of the vending machine just keeps collecting coins and instead of delivering coffee it becomes quiescent. This faulty behaviour is reflected with the quiescence transition in the product LTS depicted in Figure 9. The test purpose for the unsafe state is determined as follows. First, the set of states leading to the unsafe state (*L2U*) is computed by searching backward from the unsafe state to the initial state. Then, the test purpose is completed with all observable transitions from the unsafe state which do not lead to fail, followed by a star labeled self-loop.

The obtained test purpose enables to assess if a given test case is able to kill a mutant via a certain unsafe state. Consider the two linear test cases *TC1* = ⟨*Coin, Ready, Cancel*⟩ and *TC2* = ⟨*Coin, Ready, Coin, Ready, Ready, Coffee, Coin*⟩; note that *obs* and *ctr* keywords are omitted. The calculation of the synchronous product $TC1 \times TP$ yields ⟨ *Coin, Ready* ⟩ which is not equal to *TC1*. Hence, the test case is not able to kill the mutant via the given unsafe state. For *TC2* the product with the test purpose yields the test case itself. Thus, *TC2* fulfils the test purpose. □

4.5. S5 - Lazy Ignorant Killing Strategy

Like S4, strategy S5 also avoids creating duplicate test cases. However, S5 further tries to minimise the size of the generated test suite. Before creating test cases for a mutated specification, S5 first checks whether any of the previously created test cases is able to kill the mutated specification. A test case *tc* kills a mutant *m* iff $\neg$(*m ioco tc*). In order to verify the violation of *ioco*, a trace of the test case where the subset inclusion of observations does not hold, has to be found as can be seen in Equation 9.

Therefore, the synchronous product modulo *ioco* of the test case tc with the mutated specification m is computed. The following rules define the operational semantics for the product calculation:

$$\frac{a \in L \cup \{\delta\} \quad tc \stackrel{a}{\Rightarrow} tc' \quad m \stackrel{a}{\Rightarrow} m'}{(tc, m) \stackrel{a}{\Rightarrow} (tc', m')} \ (1) \qquad \frac{a \in L_I \quad tc \stackrel{a}{\Rightarrow} tc' \quad m \stackrel{a}{\not\Rightarrow} m'}{(tc, m) \stackrel{a}{\Rightarrow} (tc', m)} \ (2)$$

$$\frac{a \in L_U \cup \{\delta\} \quad tc \stackrel{a}{\not\Rightarrow} tc' \quad m \stackrel{a}{\Rightarrow} m'}{(tc, m) \stackrel{a}{\Rightarrow} (fail, m')} \ (3)$$

Rule (1) describes the case where tc and m synchronise on common events and proceed to their next state. Due to the input-enabledness assumption the mutated specification has to accept all input events in every state. This is realised by making every state of m input-complete by adding self-loops for the remaining input events, see Rule (2). Rule (3) states that the test case reaches a fail state when the mutant produces an output event which the test case cannot follow. Given these rules

DOI: 10.1002/stvr

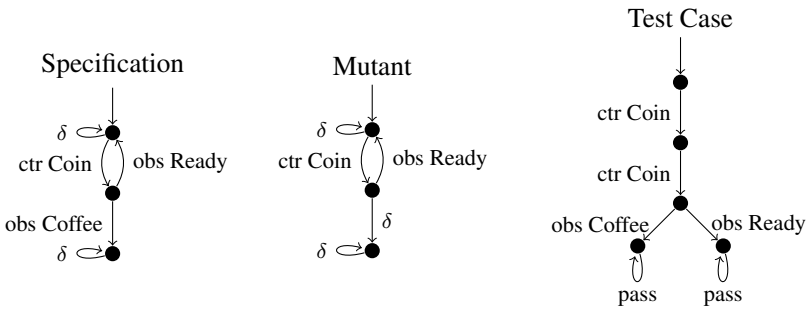

Figure 10. The first LTS shows the specification of a coffee vending machine, the second LTS represents a mutated version where no coffee is delivered, and the third LTS depicts a test case which kills the mutant.

a mutant is killed when a fail state can be reached, i.e.,

$$\exists \sigma \in (L \cup \{\delta\})^* \bullet (tc, m) \stackrel{\sigma}{\Rightarrow} (fail, m').$$

The check is done without calculating the full *ioco* product between the specification and the mutant, which is the first difference to S4. Strategy S4 always calculates the full *ioco* product. In S5, if an existing test is able to kill the mutant, the test suite is considered strong enough and no new test cases are generated for the mutant. This is the second difference to S4 as S4 will only skip test case generation for covered unsafe states, while S5 never generates any additional test case for a killed mutant. Due to this minimisation, strategy S5 is sensitive to the ordering of mutants. Strategy S5 starts with an empty test suite.

Example 6. Consider the test case and the mutated specification in Figure 10. In the following, the stepwise execution of the test case on the mutant leading to a fail verdict is explained. The first event (Coin) is processed with Rule 1 since both, the test case and the mutant can execute the event. The second coin is inserted into the machine without a prior observation of the Ready signal. This unspecified input event is handled by Rule 2. Here, the tester moves to its next state whereas the mutant remains in its current state. Finally, the test case expects the observation of either *Coffee* or the *Ready* event. However, the mutant stays quiescent which leads to a fail state according to Rule 3. By showing that there exists an execution trace of the test case to a fail state it is ensured that the test case is able to kill the mutant. □

4.6. S6 - Random First Killing Strategy

Strategy S6 works like S5 but instead of starting with an empty test suite, S6 uses randomly generated test cases to start with. In effect, S6 is a combination of random (S7) and fault-based testing (S5). The idea is to kill as many mutants as possible with a few random test cases. For the surviving mutants additional test cases are generated according to approach S5. Random test cases are easy to generate while the computationally more costly mutation-based test cases are generated only for a subset of all mutants.

4.7. S7 - Random Killing Strategy

This strategy uses a random selection approach in order to generate test cases. Put differently, S7 is not a fault-based test case generation strategy and included for evaluation purposes only.

Generating random test sequences is rather easy and computationally not very demanding. The basic idea is to record the events that should be used to stimulate the system as well as all events that can be observed from the system while doing a random walk on the specification. Since the random test cases are linear (cf. 3.3.1), inconclusive verdicts have to be added where appropriate. More precisely, such a verdict is given, if the SUT produces a correct observation different from the randomly selected expected observation. Obviously, an inconclusive verdict does not kill a mutant. The following example demonstrates this.

DOI: 10.1002/stvr

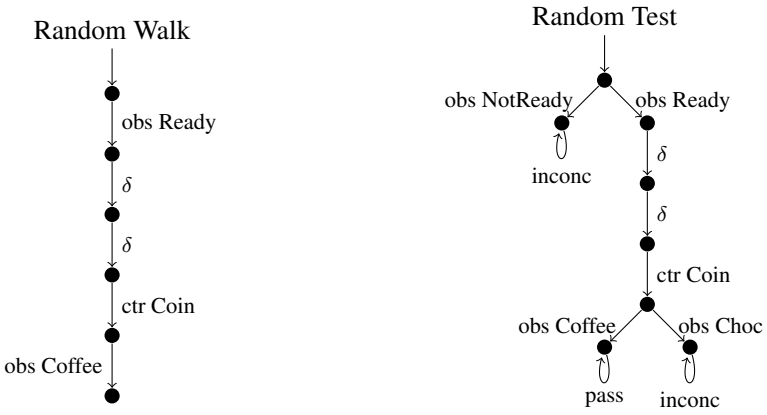

Figure 11. The LTS on the left shows a possible random walk through the specification shown in Figure 6. The LTS at the right shows the test case obtained by adding test verdicts to the random walk sequence.

Example 7. Given the vending machine specification from Figure 6, a random walk could yield a sequence like the LTS at the left of Figure 11. After adding test verdicts, a non-adaptive random test like the LTS at the right of Figure 11 would have been obtained. Notice the differences between this test case and the one produced by approach S1: the observable *Coffee* is needed for a pass verdict and the unsafe states do not have to be considered. □

4.8. S8 - Purposeful Killing Strategy

The last strategy serves the purpose of comparing the mutation-based test case generation methodology with manual test case generation. More precisely, S8 uses manually created test purposes together with a system specification and the tool TGV [10] to generate test cases.

In TGV a test purpose is a deterministic IOLTS (IOTS) that is equipped with two sets of sink states, called *Accept* and *Refuse*. The former defines the pass verdicts and at least one accept state must be present while the latter is used to limit the exploration of the graph during test case generation. A test purpose has to use the same alphabet as the specification transition system. TGV uses transitions labeled with a star ("*") in test purposes as a shorthand for "otherwise". Test cases are then generated "on-the-fly" from the synchronous product of the specification TS and the test purpose.

Example 8. Given the specification of the car alarm system (see Section 2), the test purpose shown in Figure 12 is needed to create a test for the *AcousticAlarm_SetOff* event. Notice that there are several different ways through the state machine to reach the event under investigation. TGV, however, will only report one particular event trace, in this case the LTS on the right of Figure 12.□

5. EXPERIMENTAL RESULTS

In the preceding sections, the theoretical basis of the model-based mutation testing framework has been discussed and eight different strategies for test case selection have been presented. This section goes one step further as empirical results of applying the different approaches to two case studies are shown. All of the experiments were conducted on a computer equipped with two 2.5 GHz quad-core processors and 30 GB RAM running a 64-bit Linux. While the test case generator does not support multi-threaded computations, it is possible to run multiple instances in parallel, each instance taking care of one mutant.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

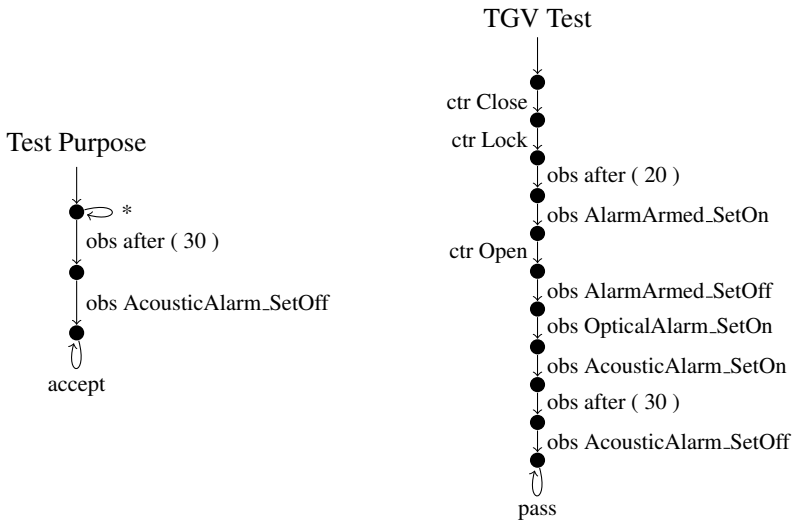

Figure 12. The LTS on the left represents a test purpose to test the *AcousticAlarm_SetOff* event of the car alarm system. The second LTS depicts the corresponding test case generated by TGV (S8).

Table II. Number of test cases for the CAS generated by the eight different approaches.

	S1	S2	S3	S4	S5	S6	S7	S8
Max. Depth	10	14	23	23	23	150 (19)	150	30
Gen. TCs [#]	16 210	302	504	129	63	11	3	9
Duplicates [#]	12 179	174	217	0	0	0	0	0
Unique [#]	3 469	110	269	123	59	11	3	9
Gen. Time [min]	188	91	23	70	23	10	0.25	-

5.1. Car Alarm System

The car alarm system (CAS) has already been presented in Section 2. It serves as a first case study to compare all eight strategies on (1) the level of test case generation by means of metrics like the number of generated test cases and the time needed for their generation and (2) on the level of test case execution by comparing the fault detection rates, i.e. the mutation scores, of the generated test suites applied on various faulty implementations.

5.1.1. Test Case Generation All fault-based strategies use a set of 76 faulty specifications (cf. Section 2.3) as an input to test case generation. Note that because the car alarm system is deterministic, all test cases are linear. As already described in Section 4, the test cases resulting from S1 and S2 have been generated via straightforward path search. The strategies S3 - S5 employ Algorithm 1. Strategy S6 generates test cases randomly and according to Algorithm 1. Table II highlights the main differences between the strategies regarding the generated test cases.

S1 - All Paths Killing Strategy For the experiments, the maximum depth for loop unfolding was chosen to be 10 ($d = 10$) and a two-times maximum visit per state per trace ($n = 2$) has been defined. As can be seen in Table II, this greedy test case generation strategy produces a very large number of tests. Also, approximately 75% of the generated tests are duplicates. From the non-duplicate test cases, another 562 tests turned out to be a complete subsequence of one of the other remaining test cases. Hence, about 3500 tests remain. The number of test cases generated by S1 has been considered as being too high for a model of this size. Hence this strategy has been given up and the tests have not been executed on the CAS implementations.

DOI: 10.1002/stvr

S2 - Target-Oriented Killing Strategy 58% of the test cases generated by S2 were identified to be duplicates and another 22 were covered as a subsequence in other test cases. Because the maximum test length has been set to 14 for this strategy the test-case generation times are not directly comparable between S1 and S2. Note that all unique test cases with a length of up to 10 that were produced by this strategy were included in the set of tests generated by the first strategy (S1).

S3 - Adaptive Killing Strategy Strategies S1 and S2 generated test cases that were restricted by a depth of 10 and 14 respectively. Strategy S3 creates theoretically unbounded test cases. Therefore, the generation time is not readily comparable with the ones of approaches S1 and S2 (but to those of S4 to S7).

S4 - Lazy Killing Strategy Compared to S3, the total number of generated test cases decreases from 504 to 129. At the same time, the amount of time spent to generate the tests increases from 23 to 70 minutes. This happens because S4 needs to compute a subset of already generated test cases that covers a maximum number of the unsafe states in the *ioco* product graph. If an unsafe state (there may be several per mutated specification) is not covered, a new test case is emitted.

S5 - Lazy Ignorant Killing Strategy This strategy brings a further reduction in the number of test cases, i.e., approximately one half, compared to S4. Like for approach S4 no duplicate test cases are generated. Since the number of unsafe states is possibly much higher than the number of mutants, the number of checks to determine if a test case kills a mutant is lower than the number of checks to determine if a test case covers an unsafe state. Hence, the generation time reduces from 70 minutes for approach S4 to 23 minutes.

S6 - Random First Killing Strategy In the experiments, the test suite was initialised with one randomly generated test case to start with. In Table II, the maximum depth not put in brackets is the depth of the randomly generated initial test case, while the figure in brackets is the maximum depth of the additionally generated tests to cover all faulty specifications.

S7 - Random Killing Strategy Random test case generation for the CAS proved to be straight forward. The only decision that had to be taken was the maximum depth and the amount of test cases to be produced. From Figure 3 it can be derived that the outer loop from State 0 over State 4 involves 14 steps at minimum. For random testing the exploration depth should be chosen much greater. Hence, the chosen depth for three generated random test cases has been 50, 100, and 150 respectively.

As can be seen in Table II the time spent generating these three random test cases is negligible when compared to the other strategies (incl. S8).

S8 - Purposeful Killing Strategy For the experiments, 9 different test purposes have been created by hand and TGV [10] was used to generate test cases. Three out of the 9 test cases check for observable timeouts (time-triggered transitions: 20, 30, 300 sec delay). Four test cases check the entry and exit actions of the states *Armed* and *Alarm*. One test case checks for the deactivation of the acoustic alarm after the timeout. One more complex test case has a depth of 30 transitions going once through the state *SilentAndOpen* to *Armed* before going to *Alarm* again and leaving after the acoustic alarm deactivation by an unlock event. Hence, each observable event is covered by at least one test case. The designer of the test purposes relied on a printout of the UML state machine.

5.1.2. Test Case Execution

SUT / Implementation Classical mutation analysis has been applied to evaluate the effectiveness of the different test suites. For this purpose the CAS has been implemented in Java according to the state machine of Figure 2. Java interfaces have been designed for both SUT and the environment

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table III. Mutations of the CAS implementation.

	Mutants	Equivalent	Redundant	Different Faults
SetState	6	0	1	5
Close	16	2	6	8
Open	16	2	6	8
Lock	12	2	4	6
Unlock	20	2	8	10
Constr.	2	0	1	1
Total	72	8	26	38

so that it was easy to connect SUT and the test driver. Note, in general it is easier to write a test driver for a simulation using the same signals as the test model, i.e. staying on the same level of abstraction, than to create a test driver for real hardware.

The interface of the SUT declares a method for each of the signals *Lock*, *Unlock*, *Close* and *Open*. Since the implementation uses simulated time, the interface also declares a tick method that is called by the environment, i.e. test driver, to tell the CAS that one unit of time has passed.

Test Driver The test driver implements the environment interface. This interface declares call-back methods for the signals *AlarmArmed_SetOn*, *AlarmArmed_SetOff*, *AcousticAlarm_SetOn*, *AcousticAlarm_SetOff*, *OpticalAlarm_SetOn* and *OpticalAlarm_SetOff*.

Abstract test cases that are provided in the ALDEBARAN format (suffix *.aut*)** are parsed by the test driver. Whenever a controllable action is found, the corresponding method of the SUT is called. The SUT can respond to the test driver by calling a call-back method. These calls are recorded by the test driver. Whenever the test driver parses an observable action, this action is compared to the oldest recorded call-back method call. If the labels match, the execution of the test case is continued, otherwise it is aborted and the test verdict *fail* is given.

When an *after* event is parsed denoting that the test driver shall wait for a specified time, the tick method is called repeatedly. Before calling the tick method for a last time, the recorded observations are checked for unspecified behaviour. Then tick is called once again and parsing of the test case is continued.

When the end of a test case is reached without detecting unexpected behaviour, the verdict *pass* is given.

Results In order to derive a set of faulty implementations, version 3 of μJava [14] has been used. All traditional mutation operators (method-level operators) have been applied. In total, μJava created 72 mutated implementations. After careful manual inspection, 8 of these mutated implementations were found to be equivalent to the original program, and another set of 26 mutants was found to be redundant because the mutants were syntactically equivalent to other mutants.†† Usually, in mutation testing one should avoid equivalent or redundant mutants. Because of that, they have been singled out and a sum of 38 (72 - 8 - 26) different faulty implementations of the CAS remain.

Further details are shown in Table III. For each method, the table lists the total number of mutated implementations, the number of mutants that turned out to be equivalent to the original implementation, and the number of equivalent pairs of mutated implementations. The methods *Close*, *Open*, *Lock* and *Unlock* are public and defined by the interface while *Set State* and the constructor (*Constr*) are internal methods. From the table, one can observe that the mutation of internal methods has a strong effect on the external behaviour since there are no equivalent mutants for these methods.

**`http://cadp.inria.fr/man/aldebaran.html#sect6`

††The high number of syntactically equivalent mutants is due to what seemingly is a bug in the LOI operator of μJava, version 3. The operator generates redundant mutants when applied to a final static Integer field of another class.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table IV. Overview of how many faulty SUTs of the CAS could not be killed by the test cases generated with the different approaches.

	S2	S3	S4	S5	S6	S7	S8
SetState	0	0	0	0	0	0	0
Close	1	0	0	0	0	0	2
Open	0	0	0	1	0	0	4
Lock	0	0	0	0	0	0	2
Unlock	1	0	0	0	0	1	5
Constr.	0	0	0	0	0	0	0
Total	2	0	0	1	0	1	13
Mutation score [%]	95	100	100	97	100	97	66
Impl. Coverage [%]	99	99	99	99	99	99	88

All 38 unique faulty CAS implementations were used to evaluate the effectiveness of the generated test cases. Of course, all tests were validated beforehand on the non-mutated CAS implementation.

Table IV gives an overview of the number of survived faulty implementations for each test case generation strategy. As can be seen from the table, the strategies S3, S4, and S6 were able to reveal all faults. The table also shows that the minimisation algorithm applied in approach S5 reduced the fault-detection capabilities. With the reduced number of test cases of S5, one mutant survived. Despite random testing (S7) did not find all faulty implementations, it proved quite effective in the given setting. Summing up, S5 and S7 still have a mutation score of 97%. The reason for the bad performance of S2 is the fact that the two surviving faulty implementations need tests with a depth of more than 14 interactions to be revealed and S2's tests are restricted to a depth ≤ 14.

The last column shows the results of running the manually designed tests (S8). Overall, 25 mutants were killed which results in a mutation score of 66%. Clearly, the figures show that in order to have a meaningful test suite, more (diverse) test cases have to be generated. Partly, this lack of diverse test cases is based on the deterministic test selection behaviour of TGV: all TGV-based tests have almost the same test sequence from *OpenAndUnlocked* to *ClosedAndLocked*, although alternative paths are possible.

Often, coverage metrics on the implementation's source code serve as a quality measure to describe the adequacy of a test suite. Therefore, the coverage on the implementation in terms of basic block coverage has been measured. The strategies S2 - S7 achieve a coverage of 99%, whereas S8 (TGV) only results in a basic block coverage of 88%. By comparing the last two rows of Table IV, it becomes obvious that a high code coverage does not automatically guarantee high mutation scores, which is the overall goal.

In summing up, the results show that the strategies are powerful in covering the implementation's source code and more importantly in detecting faults. However, the depth of the conformance analysis is critical as a too shallow bound for the exploration depth results in missing test cases and, in this example, in undetected faults. The results also show that the 3500 test cases of the first strategy were too many by far: for the given model mutations, one path per mutation is sufficient to detect all faults, provided this path is long enough. Finally, the combined strategy (S6) proved to be a nice trade-off between generation time and effectiveness.

5.2. A Second Case Study: a Wheel Loader

In a second experiment, the model of an ECU (Electronic Control Unit) controlling the arm and the bucket of a wheel loader served as a case study. Figure 13 shows the movable arm/bucket of a Lego model next to the ECU that sits in the back. The system under test (ECU) was implemented by the industry partner RE:Lab on a Freescale i.MX35 processor running Linux.

The principle system architecture of this demonstrator is as follows. The ECU receives signals from an ISOBUS (CAN) network and controls the actuators of the bucket/arm. Information about the current status of the system is displayed both on a directly connected TFT display and an

DOI: 10.1002/stvr

Figure 13. A Lego model of a wheel loader used for test case execution in the MOGENTES project.

ISOBUS terminal in the cabin. Inputs to the ECU come from a joystick that is connected to the ISOBUS.

Compared to the car alarm system, in which three core requirements could easily be identified, the requirements of the wheel loader were much more complex. Important behavior that had to be tested was providing suitable values for the actuators fulfilling the physical constraints, error management, and general timing-related properties. This model proved to be a challenge as different low-level bus-related events had to be modeled in detail (i.e., integer-parametrized events) while at the same time it also showed a high degree of parallelism (i.e., interleavings of event sequences) and complex time-out behavior.

5.2.1. Tool Improvements The following improvements were made to the mutation-based test-case generation tool so that it can handle the additional features found in the second case study.

First, the transformation from UML to LTS has changed slightly with respect to time events. Now, time is modeled by adding a parameter (the first one) to each action. This parameter denotes the time that has passed as integer. Hence, the explicit *after*-action has been hidden. If an action is observable, then the SUT waits exactly the specified time before the observable is shown. Otherwise, if an action is controllable, then the parameter denotes the time that has passed between the occurrence of the last action and the occurrence of the input from the environment. The reason for this change is the existence of so-called *mixed states* in the LTS of the wheel-loader models. A mixed state has both controllable and observable events enabled. In such mixed states, it is more consistent to link the waiting time before a controllable action to this action, than to introduce a new observable action. In the car alarm system, each time event was followed by observable actions only. As a consequence of removing the after-actions, the exploration depths between the two case studies are not exactly comparable. Notice that a trace through the system now closely resembles the usual notion of a timed trace where each action is represented as tuple of time-delay and action.

Second, a change has been made to the test case generator *Ulysses*. The original strategies S1–S5 relied on a breadth-first search up to the given search depth. This worked especially fine for

DOI: 10.1002/stvr

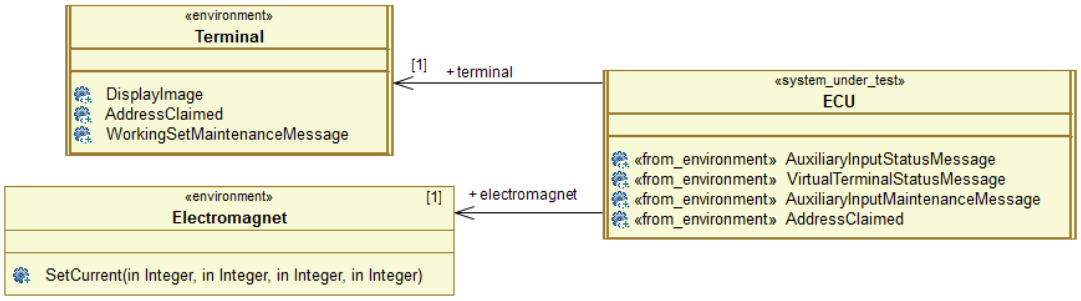

Figure 14. The testing interface for the bucket control of the wheel loader.

strategies S3–S5 when handling models as small as the car alarm system, as the longest path in the LTS was smaller than the search depth. When working with larger models it is necessary to adapt these strategies as follows: now, the exploration of all alternative branches is stopped at the depth of the first found fail-state. Previously, alternative branches without fail-states were fully explored up to the given search depth. Unfortunately, this optimization does not help with equivalent mutants.

5.2.2. UML Models Figure 14 shows the test interface for the wheel loader. The ECU may receive several different ISOBUS messages from its environment, like the *AuxiliaryInputStatusMessage* that is generated by the Joystick. Practically all these messages carry several integer parameters. The environment of the ECU is split into two classes modeling the electromagnets used to control the bucket/arm and a terminal that is used for displaying user information like the current error state, joystick position etc. All signal receptions in the environment classes are outgoing signals as seen from the ECU. For example, the ECU is responsible for setting the current that flows through the electromagnets and controls some valves on the actuators on the arm/bucket of the wheel loader.

To cope with the complexity of the SUT, two partial state-machine models have been built. The aim of partial models is to capture different functional aspects of a system. The first partial model *X_error* includes all of the error handling functionality, the second partial model *Extremes* captures the so-called positive behavior dealing only with valid data and correct timing behavior.

The state machine diagram of the model containing the error handling is shown in Figure 15. As can be seen, it makes heavy use of hierarchical states, orthogonal regions, timeouts, and constrained transitions.[‡‡] Once initialized, several activities are started: error management, input reception, output generation, and additional watch-dog-like tasks are all being run in parallel. In this model, input reception and output generation are separated into two different regions as outputs need to be generated at a fixed rate while inputs arrive at a variable rate.

However, when modeling the system without error handling, these two regions can be merged, leading to a model which is simpler. Figure 16 shows the state machine of the partial model, which focuses solely on the calculation of the outputs for setting the current by which the electromagnets are powered.

5.2.3. Generated OOAS Test Models Translating the UML models into the OOAS language leads to a model consisting of 122 (*X_error*) and 119 actions (*Extremes*) respectively as can be seen in Table V. These include the four input actions to the ECU, the three output actions to the terminal and one output action to the electromagnet as can be seen in the specification of the testing interface in Figure 14. All other actions are internal.

Figure 17 shows a sample test case. The first action belongs to the bus initialization phase. The next three actions correspond to the initial outputs generated by the ECU. After 25 time units, all

[‡‡]While the state machine could be split into separate state machines per orthogonal region, the current UML-to-OOAS transformation only supports one state machine per class.

DOI: 10.1002/stvr

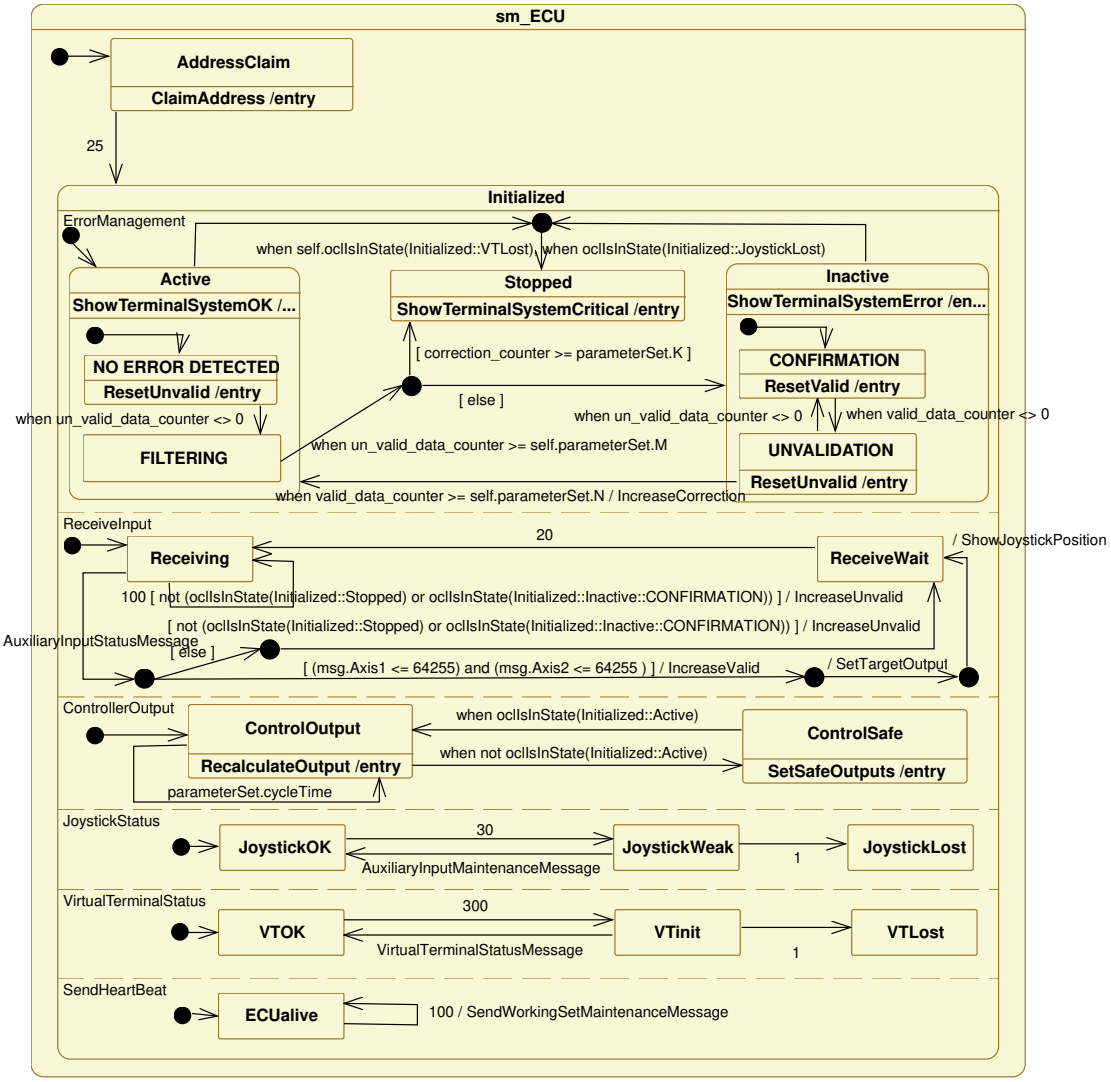

Figure 15. The UML state machine of the model describing the error handling (*X_error*) of the bucket control of the wheel loader.

four currents are set to 0 and the images stating that the ECU is in a healthy state and neither bucket nor bucket arm are moving are sent to the terminal. Finally the first input can be processed. The value 62649 represents the most right position on the X axis, the value 32127 represents no deflection in the Y axis, the so-called *rest position*. The desired behavior is a rotation towards the bottom but no movement of the bucket arm. The test case first checks, whether the image with id 56, corresponding to the rightmost joystick position is sent to the terminal and that after 10 time units the first port of the first electromagnet is powered with 50 mA.

DOI: 10.1002/stvr

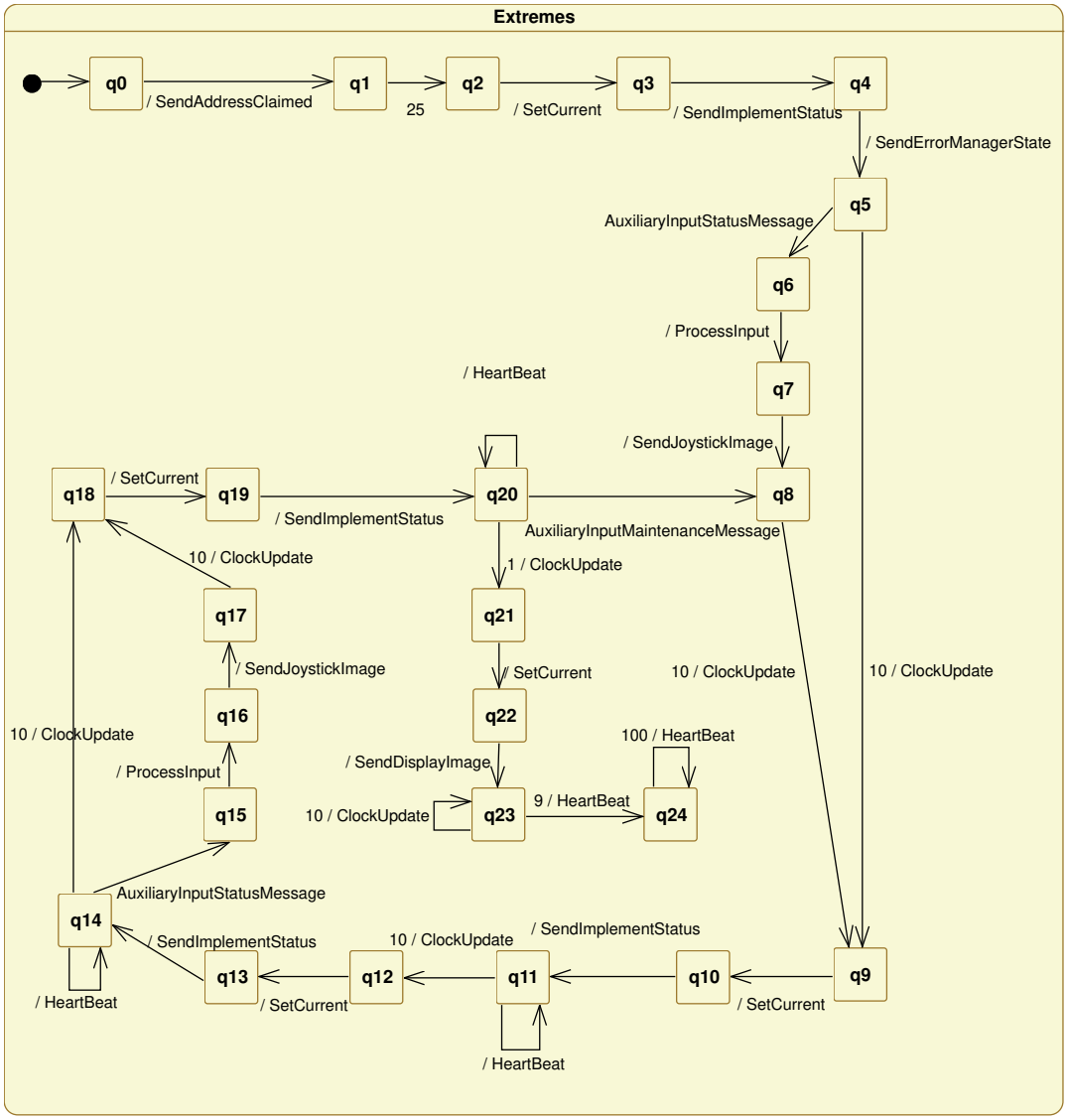

Figure 16. The UML state machine of the partial model (*Extremes*) of the wheel loader.

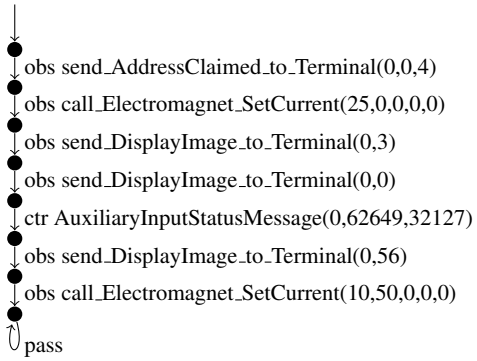

Figure 17. A sample test case for the wheel loader case study.

Table V. Important characteristics for both versions of the wheel loader model.

Model Metrics	X_err	Extr
Number of actions	122	119
Max. depth for *ioco* check	35	25
Mutants [#]	58	192
Equivalent mutants [#]	35	106
Mutants causing a livelock	0	8

DOI: 10.1002/stvr

Experiments [15] have shown that the tool *Ulysses* is very sensitive about the ranges of parameter values. In this case study, this has to be considered twice. First, one needs to take care of choosing the right time unit. Time events in this model can be as high as 3000 milliseconds. While a range from 0 to 300 for a parameter works perfectly fine with Ulysses, a range of 0 to 3000 can render the enumeration of the actions to a bottleneck of the exploration. To avoid this issue, 10 milliseconds has been chosen as a time unit for this model. So a time value of 10 denotes 100 milliseconds in reality. Second, attention has to be given to the parameters of the action *AuxiliaryInputStatusMessage*. This action models the message from the joystick to the ECU containing the deflection of both axes and it is parameterized with two integers ranging from 0 to 65355. The tool *Ulysses* would enumerate all these values, branch execution on each step and run into the state space explosion problem. In order to cope with the complexity of the fault-based strategies on this use case, the input values of the partial models have been partitioned into equivalence classes.

Model X_error The *X_error* model is complete regarding the state machine, but rigorously constricted in its input parameters. In order to be able to explore the model deep enough, only two values for the joystick deflection have been considered: One valid joystick deflection value representing the leftmost joystick deflection and one invalid value representing an error reported by the joystick. Since the Y axis in this model is fixed to the neutral position and both the valid and the invalid value are in the parameter of the X axis, this model is referred to as *X_error*.

Model Extremes The second partial model *Extremes* captures the calculation of the outputs to the actuators in a way that the tool *Ulysses* can reach interesting depths in reasonable time. In this model, nine possible deflection values for the joystick deflection have been chosen. They include eight deflections representing eight different directions and additionally the neutral position being in the middle. Since no value representing an error value is included, most parts of the error handling can be ignored. Only if a timeout occurs, the system is defined to switch to an error state, in which it halts forever. Thus all orthogonal regions were eleminated by merging their contents into one remaining region. While this does not solve the underlying problem of exponential branching of execution paths, it allows to explore the model deep enough and enables the generation of meaningful test cases.

5.2.4. Mutations When using large models the number of generated mutants becomes important, as the processing of one equivalent mutant may take up to three hours (see Table VII). Hence, the total number of possible mutants given the set of tool-supported mutation operators has been considered too large and only the following three mutation operators have been selected:

1. Increase literal integer values in effects by one
2. Increase literal integer values in guards by one
3. Increase time in time events used as triggers by one

Together these mutation operators capture the fault model of so-called off-by-one errors. The assumption is that more course-grained faults, like e.g., the removal of a transition, will be covered by these fine-grained faults. For program mutation this effect is known as “coupling effect” and has been investigated e.g. by Offutt[16]. Although this effect has not yet been shown for UML state machines, the authors conjecture that it holds. The intuitive reason is that in order to reach the unsafe state in the model, the tester has to steer the system along some path while always checking the SUT’s behavior for conformance. Hence in order to reach a state where the off-by-one error reveals itself to the environment, the system will have executed a sequence of inputs/outputs and traversed a set of transitions, covering, e.g. the “guard false” mutation operator to some extent. The conjecture is also corroborated by the results presented for this case study. In general, however, relating fault coverage with more classical notions of coverage and finding an ordering of mutation operators with respect to effectiveness remains future work.

A total of 58 mutants (model *X_error*) and 192 mutants (model *Extremes*) has been obtained using these three mutation operators.

DOI: 10.1002/stvr

Table VI. Number of generated test cases and mutation scores for strategies S3–S7.

Strategy	S3		S4		S5		S6		S7	
Model	X_err	Extr	X_err	Extr	X_err	Extr	X_err	Extr	X_err	Extr
Gen. test cases [#]	31	356	17	150	14	60	13.3	28.3	3	3
Mutation score [%]	33.76	66.62	33.76	66.62	33.61	65.34	34.63	65.64	27.33	56.26
Total mut. score [%]	74.12		74.12		72.84		73.04		60.14	

Table VII. Run-times t in minutes for generating test cases according to strategy S3–S6.

Strategy	S3		S4		S5		S6	
Model	X_err	Extr	X_err	Extr	X_err	Extr	X_err	Extr
Avg. t for equivalent mutant	161.8	35	162.3	35	162.7	36	156.8	36.8
Total t for equivalent mutants	5663	3711	5679	3709	5696	3812	5487	3849
Avg. t for mutant causing livelock	0	6.2	0	6.2	0	7.2	0	5.6
Total t for mutants causing livelock	0	50	0	50	0	58	0	40
Mutants triggering TC [#]	23	78	9	41	6	23	5.3	9.3
Avg. t for mutant triggering TC	7.3	2.1	19.6	5.9	27.7	3.4	36.2	6.8
Total t for mutants triggering TC	168	167	176	241	166	79	181	410
Mutants killed by existing TC [#]	–	–	14	37	17	55	17.6	71
Avg. t for mutant killed by TC	–	–	0.1	1.7	0.1	0.3	0.4	0.7
Total t for mutants killed by TC	–	–	2	63	1	15	25	48
Total t for TC gen	5831	3928	5857	4063	5863	3964	5693	4347

5.2.5. Test Case Generation Strategies S3–S7 have been applied to the two models of the wheel loader. For strategy S7, which uses a random exploration to generate a test suite, the same parameters (three test cases with search depth 150 each) have been used as in the car alarm system case study. The values reported in TableVII are the mean values of three runs.

For strategy S6 that uses these test cases as initial test suite, the test case generation process has also been repeated three times. Hence, the reported values for strategy S6 are also mean values. In total, twelve independent test suites using fault-based test case generation approaches have been generated.

Table V lists the metrics of the models which are independent of the strategy: For the *Extremes* model, a search depth of 25 has been chosen, for the *X_error* model, a search depth of 35 was used. Up to these exploration depths, 106 of the 192 mutants of the *Extremes* model and 35 of the 58 mutants of the *X_error* model had to be considered equivalent, i.e., they are input-output conform to the original model with respect to the given exploration depths.

Since it might happen that mutated specifications run into infinite loops of internal actions, a so-called *livelock*, there is a timeout of 5 minutes for each step between two visible actions during test case generation. Eight of the *Extremes* mutants caused a livelock, but none of *X_error*.

The main difference among the fault-based test case generation strategies S3–S6 lies in the number of generated test cases. Table VI lists the number of test cases for each application of a strategy for both test models. Compliant with the results of the car alarm system case study the number of generated test cases decreases as the strategy gets more sophisticated.

For the *X_error* model the number of generated test cases ranges from 31 using the strategy S3 to 14 using strategy S5 and an average of 13.3 using strategy S6. Hence, by using a combination of random and fault-based test case generation one is able to shrink the size of the generated test suite to less than half compared to a test suite created by a strategy that does neither of these optimizations.

For the *Extremes* model the number of generated test cases ranges from 356 using the strategy S3 to 60 using strategy S5 and only 28.3 on average when using strategy S6. Hence, the size of the test suites can shrink to less than one tenth.

In contrast to the significant reduction of the number of test cases, the total run-times for the test case generation process for each strategy stays almost constant. Table VII shows the run-time of the purely fault-based strategies as well as the run-times of the combined strategy S6. The

DOI: 10.1002/stvr

Table VIII. Details for three runs of both versions of the wheel loader model generated by strategy S6.

Test Case Generation with S6	Extremes			X_error		
	Rand1	Rand2	Rand3	Rand1	Rand2	Rand3
Generated test cases (excl. random) [#]	21	28	36	13	14	13
Mutants triggering a new test case [#]	7	12	9	5	6	5
Mutants killed by existing test cases [#]	79	65	69	18	17	18

total generation time for one test suite using strategies S3–S6 ranges from at least two days (3928 minutes) to approximately four days (5863 minutes), depending on which model is used.

The table also presents the time spent by mutant-type. The first two rows show the time spent processing equivalent mutants: if the non-conformance between the original and the mutant cannot be shown within the given depth, processing just one mutant takes from 35 to 163 minutes. This is independent of the strategy and responsible for the largest portion of time spent. Hence, equivalent mutants are the main reason, why the total time spent on test case generation does not differ much among the presented strategies.

The next two rows show how much time is spent on processing mutants where test case generation fails due to a livelock. As already presented in Table V this only happens in the *Extremes* model. It can be seen that the livelocks in the mutants occur at the beginning of the exploration, therefore comparatively little time is spent on processing these mutants.

The number of mutants triggering a new test case depends on the strategy. In strategy S3, all mutants that are neither equivalent up to the search depth nor fail because of a livelock trigger the generation of new test cases. The average time needed for processing such a mutant highly depends on how sophisticated the killing approach is. For example, strategy S3 generates many short test cases, which lowers the average generation time. In the more sophisticated strategies S4–S6, this is often suppressed by the check whether an existing test case can already kill the mutant.

The last three rows show what happens if a mutant is killed by an existing test case: The average time to kill a mutant is comparatively low. For the *Extremes* model, which has a larger range for input values that need to be enumerated, it is below two minutes. For the *X_error* model it is even faster taking less than half a minute on average.

Test case generation using a random walk is comparatively fast. The generation of each test suite containing three test cases took between two and three minutes.

Strategy S6 (the combination between random and fault-based test case generation) has been repeated three times. Table VIII presents the associated data in detail.

5.2.6. Test Case Execution

SUT / Implementation Like in the first case study, a simulation of the ECU has been implemented in Java. The source code consists of approximately 500 lines. It is based on the state machine of Figure 15, which includes the behavior of the state machine presented in Figure 16. In fact, test-driven development has been used. First a large set of random tests has been generated and then one test case after another has been implemented until all tests passed. In addition, the implementation had to pass all generated test cases of strategies S3–S7.

The design is similar to the implementation of the car alarm system and interfaces for both SUT and the environment have been defined. The environment contains methods corresponding to the signals of both, terminal and electromagnet. The interface of the ECU contains a method for each signal and additionally one *tick* method for modeling time.

Test Driver The test driver is built analogously to the test driver of the car alarm system. Again, it implements the Java interface of the environment, thus observing signals sent to Terminal and Electromagnet. Most parts of the code implementing the test driver for the car alarm system have been reused. Some parts had to be extended because of the occurrence of "mixed states" and the new notion of time.

DOI: 10.1002/stvr

Results Again the μJava [14] tool has been used and all method-level mutation operators have been applied. This yielded 1511 mutated implementations. After testing, the surviving mutants have been inspected manually to see whether they were equivalent or the injected fault was too subtle for the generated test cases to find. From a total of 345 to 532 (depending on the strategy) surviving mutants, 178 were found to be equivalent. Within this set, 80 equivalent mutants arose from a post-increment/decrement of the last occurrence of a local variable. Redundant clauses in if statements, which made the code more readable, were the reason for another 39 equivalent mutants. Further 33 equivalent mutants arose from swapping comparison operators that behave the same if the variable only stores positive integers. One integer variable is used as enum. Exchanging the comparison operator "equals" to "greater equals" for the largest value and "smaller equals" for the smallest value, leads to 8 equivalent mutants. In four cases, a variable was increased or decreased in a way that the function using it still returned the same value. Another four equivalent mutants arose by negating the integer value zero and the final two equivalent mutants changed the value of a variable right before it was set to the correct value.

Table VI gives an overview of the percentage of killed mutated and non-equivalent implementations for the test suites. This ratio is known as *mutation score*. In addition, the total mutation score of the strategies when combining the test sets from both model versions is reported. For those strategies including random test case generation, the test suites were generated three times. The reported values for S6 and S7 are the respective mean values.

6. DISCUSSION

In the following the results gained from the two case studies are discussed. It is structured according to the original questions stated in Section 1.

What is the best strategy to select test cases? The car alarm system case study has been used to filter out strategies that will not scale to larger models. As expected, S1 results in too many test cases. Like S1, strategy S2 generates a high number of test cases even for a relatively low exploration depth. Also, S2 is not applicable to non-deterministic models. The effectiveness of S8 highly depends on the skills of the test engineer. The design of a set of test purposes covering all essential aspects of a model is non-trivial and the relatively low mutation score of S8 for the CAS has been discouraging. For these reasons S1, S2, and S8 are not included in the second case study. Staying with the CAS, the adaptive test cases of S3 achieve the maximal mutation score but the number of test cases is too high. Strategy S4 reduces this number and keeps the high mutation score but the generation time increases. In order to decrease both, the number of test cases and the generation time, S5 has been developed. However, when applying this strategy the mutation score suffers slightly. Therefore, S5 is combined with random testing resulting in S6, which can reestablish the high mutation scores of S3 and S4.

In the wheel-loader case study strategies S3 and S4 also deliver the best mutation scores. In contrast to the CAS, their run-times are almost equivalent. As stated earlier, this is caused by the large number of equivalent model mutants. Strategies S5 and S6 dramatically reduce the number of test cases while the mutation score keeps nearly stable. In contrast to the CAS, strategies S3 & S4 achieve a better mutation score than strategy S6. Table VI shows that this is due to the *Extremes* model. However, given the large number of test cases of S3 (356 test cases) and S4 (150 test cases) for the *Extremes* model, S6 is still preferable in practice.

Both case studies show that pure random testing (S7) is achieving a relatively high mutation score with only a few very long test cases. However, adding the test cases of an exact conformance analysis (S6) still increases the mutation score.

Is mutation testing better than random testing? The answer depends on the requirements. Random test case generation is fast and quickly reaches a high mutation score. Hence, these tests are well suited during the development of the system. They have been used in the test-driven

Prepared using stvrauth.cls
DOI: 10.1002/stvr

development of the Java implementation for the wheel loader case study. However, with random testing there is no control about which areas of a model are explored. Therefore, many (long) tests might be needed to cover certain faults. In some environments this is perceived as too expensive. For example, in embedded systems testing with timers causing long waiting times, test execution is costly. In this case, redundant tests should be avoided.

In relation to mutation testing, randomly generated tests have no connection to fault models. Hence, it is not easy to decide how many random tests are needed. Therefore, the authors of this work propose the combined approach S6, where first a small set of random tests is generated that will find the most trivial bugs. Then for the more subtle bugs the missing test cases are systematically generated by mutation analysis.

How efficient are the presented test case generation strategies? The wheel loader case study shows that compared to random testing the presented mutation-based test case generation strategies are rather expensive when considering run-time. The reasons are the conformance checking and the number of mutants to be analyzed. Since the conformance check has exponential run-time, the maximum exploration depth needs to be limited. In case of non-conformance this works reasonably well. For example, in S5 the time to process a non-equivalent mutant for generating a test case is 27.7 min (Table VII, *Extremes* model) or 3.4 min (Table VII, *X_error* model). However, if the mutant is equivalent (up to the maximum depth), then it can take hours before the next mutant can be analyzed. Hence, we pay for equivalent mutants in terms of test case generation time.

How severe is the equivalent mutants problem? From the discussion of the previous question it can be seen that the problem affects the current technique. With an equivalent mutant, a full equivalence check has to be performed up to a certain depth. Since the problem is NP-complete, the exploration depth is limited. The wheel loader case study demonstrates this.

Do partial models help? Yes! For testing different functional aspects of the wheel loader case study, two partial models have been developed. The *X_error* model enables the generation of test cases long enough to test the error handling functionality (depth 35). Hence, this model is designed for deeper exploration. In contrast, the *Extremes* model allows for a wider range of input values to test the main functionality (depth 25). The combination of the generated tests results in higher mutation scores (Table VI). The same effect can be noticed in the pure random testing strategy S7. Combining the random tests from both models increases the total mutation scores (Table VI).

Does the combination of random testing and mutation testing help? Certainly! The data shows that the combined approach S6 has a higher mutation score than the pure strategies S5 and S7. Although, the mutation score only increases slightly, the combined approach S6 provides more trust in the testing set: the qualitative advantage of S6 is that it gives a guarantee that all modeled faults are being covered. What the random tests do not cover, the conformance checker will provide.

Given a set of faulty implementations, can all known bugs be found? In theory, only the detection of modeled faults can be guaranteed. In practice, it is possible to find a large set of additional faults. In the CAS, three of the fault-based strategies achieved 100% mutation score. In the wheel loader case study it was possible to kill around 74% of the faulty implementations with only one fault model (off-by-one errors).

7. RELATED RESEARCH

This is not the first work focusing on generating test cases from UML state machines. In the 1990s, Offutt and Abdurazik presented a tool for test case generation from UML state machines [17]. Gnesi et al. [18] define the semantics of a subset of UML state charts in the form of transition systems with

Prepared using stvrauth.cls
DOI: 10.1002/stvr

transitions labeled by input/output pairs. For this kind of models, they give a formal conformance relation similar to *ioco* and present an algorithm to automatically derive test cases. Seifert et al. [19] define so-called Compact Semantic Automaton (CSA) in order to efficiently represent the semantics of a UML state machine. From CSAs, they derive traces of observable events which serve as input for the test case generation algorithm defined by Tretmans [5]. Fröhlich et al. [20] systematically transform use cases into UML state machines and generate test suites with a given coverage level by applying AI planning methods. Various commercial tools support model-based testing with UML state machines [21]. However, none supports mutation testing.

Model-based mutation testing was first used for predicate-calculus specifications [22]. Later, Stocks applied mutation testing to formal Z specifications [23] without automation. Due to model checking techniques, full automation became feasible. By checking temporal formulae that state equivalence between original and mutated models, counter-examples are generated, which serve as test cases [24]. In contrast to this state-based equivalence test, the presented technique checks for input-output conformance allowing non-deterministic models. The idea of using an *ioco* checker for mutation testing comes from Weiglhofer, who tested against Lotos specifications [25]. To the knowledge of the authors, their work is the first to apply this technique to UML state machines. Indeed, there has been previous work on mutation testing of UML state machines but in terms of model validation [26]. It seems that the tool presented in this work is the first to actually generate test cases from mutated UML state machines.

Hierons and Merayo [27] dealt with mutation-based test case generation for probabilistic finite state machines. The work presents mutation operators and describes how to create input sequences to kill a given mutated state machine. Briand et al. [28] investigated state-based testing of classes. Their work presents results for testing state charts and evaluates their approaches on a set of mutated implementations.

Black et al. argue in their work on automating model-based mutation testing with the model checker SMV that “A practical system must extract state machines from higher level descriptions such as SCR specifications, MATLAB state-flows, or UML state diagrams“ [29]. The authors of this work agree and see one contribution of this work in satisfying this requirement.

Regarding time, the interleaving semantics of parallel running timers like UPPAAL [30] is not yet supported.

8. CONCLUSION

In this article a detailed analysis of a new model-based mutation testing technique has been presented. Different test generation strategies have been discussed in detail and were evaluated in two case studies. A small but non-trivial industrial example served as a motivating example. The second, larger case study should confirm whether the testing methods are practical for realistic systems. Hundreds of thousands of test sequences were executed on the implementations under test. The results of the experiments have answered seven research questions.

The results show that the proposed strategies for model-based mutation testing are expensive. Equivalence checking is costly and the equivalent mutants problem reduces the efficiency of the technique. This was pointed out in a recent survey on mutation testing by Jia and Harman:

> “One barrier to wider application of Mutation Testing centers on the problems associated with Equivalent Mutants. As the survey shows, there has been a sustained interest in techniques for reducing the impact of equivalent mutants. This remains an unresolved problem.” [31]

However, it could be shown that by applying partial models and adding random strategies the situation was improved considerably. These techniques need further evaluation.

One important conclusion is that by adding mutation testing to a random testing strategy approximately the same number of bugs were found with fewer test cases. Furthermore, by combining the two strategies, knowledge can be gained about which of the modeled faults are

DOI: 10.1002/stvr

covered by the test cases. This knowledge about the own test suite can be considered as an important advantage of this approach.

Despite the equivalent mutant problem, the authors of this work see many directions for further improvement. One obvious point for improvement is the conformance checker *Ulysses*. It is basically an explicit model checker that can be improved in many ways. Currently, the authors are working on a symbolic conformance checker for action systems based on constraint solving [15, 32, 33, 34]. First experiments have shown promising results. By now, the symbolic tool has been applied to the car alarm system. The refinement checks for 207 mutants require less than three seconds, whereas *Ulysses*, the explicit *ioco* checker used in this work, spends 68 seconds for the same set of mutants [33]. The symbolic tool is still under development and does not yet support the full input language required for the translation from UML state machines to action systems. Hence, it could not yet be applied to the very same case studies presented in this article.

The authors also pursue a second direction: in a further refinement of strategy S6 the authors attempt to combine directed random exploration and mutation based test-case generation ”on-the-fly”. The idea is that during the exploration of the system, mutations – which are close to the current state – are dynamically inserted and conformance is checked. This strategy promises to remove the need to pre-compute a huge set of model-mutants. It also fixes the expensive coverage analysis, currently run as a kind of after-thought, as the information about the current location in the LTS is retained across different applications of mutation operators. In addition and since the mutation is guaranteed to be local, the depth of the conformance check can be bounded as well and so the equivalent mutant problem is further mitigated.

There is a lot of work to be done, since

> “Most work on Mutation Testing has been concerned with the generation of mutants. Comparatively less work has concentrated on the generation of test cases to kill mutants.” [31]

The presented work in this article addressed this gap in research.

ACKNOWLEDGEMENTS

Research herein was funded by the EU FP7 project ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems (MOGENTES). Research herein was also partly funded by the Austrian Research Promotion Agency (FFG), program line “Trust in IT Systems”, project number 829583, TRUst via Failed FALsification of Complex Dependable Systems Using Automated Test Case Generation through Model Mutation (TRUFAL) and by the European Artemis Joint Undertaking project MBAT, Combined Model-based Analysis and Testing of Embedded Systems, ART Call 2010: 269335. The requirements for the CAS example were provided by Ford. The requirements of the bucket controller came from RE:Lab.

REFERENCES

1. Bernot G, Gaudel MC, Marre B. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal* 1991; **6**(6):387–405.
2. Gaudel MC. Testing can be formal, too. *6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT ’95)*, *Lecture Notes in Computer Science*, vol. 915, Springer, 1995; 82–96.
3. Hamlet RG. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering* 1977; **3**(4):279–290.
4. DeMillo R, Lipton R, Sayward F. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* April 1978; **11**(4):34–41.
5. Tretmans J. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools* 1996; **17**(3):103–120.
6. Krenn W, Schlick R, Aichernig BK. Mapping UML to labeled transition systems for test-case generation – a translation via object-oriented action systems. *Formal Methods for Components and Objects (FMCO) 2009*, *Lecture Notes in Computer Science*, vol. 6286, Springer, 2009; 186–207.
7. Aichernig BK, Brandl H, Jöbstl E, Krenn W. UML in action: a two-layered interpretation for testing. *ACM SIGSOFT Software Engineering Notes* 2011; **36**(1):1–8. UML&FM 2010: Third IEEE International workshop UML and Formal Methods.
8. Aichernig BK, Brandl H, Jöbstl E, Krenn W. Model-based mutation testing of hybrid systems. *Formal Methods for Components and Objects (FMCO) 2009*, *Lecture Notes in Computer Science*, vol. 6286, Springer, 2010; 228–249.
9. Brandl H, Weiglhofer M, Aichernig BK. Automated conformance verification of hybrid systems. IEEE, 2010; 3–12.

DOI: 10.1002/stvr

10. Jard C, Jéron T. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)* 2005; **7**(4):297–315.
11. Aichernig BK, Brandl H, Jöbstl E, Krenn W. Efficient mutation killers in action. *4th International Conference on Software Testing, Verification and Validation (ICST 2011)*, IEEE, 2011; 120–129.
12. Tretmans J. Model based testing with labelled transition systems. *Formal Methods and Testing*, *Lecture Notes in Computer Science*, vol. 4949, Springer, 2008; 1–38.
13. Back RJ, Kurki-Suonio R. Decentralization of process nets with centralized control. *Distributed Computing* 1989; **3**(2):73–87.
14. Ma YS, Offutt J, Kwon YR. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability (STVR)* 2005; **15**(2):97–133.
15. Aichernig BK, Jöbstl E. Towards symbolic model-based mutation testing: Combining reachability and refinement checking. *7th Workshop on Model-Based Testing (MBT 2012)*, *EPTCS*, vol. 80, 2012; 88–102.
16. Offutt J. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology* Jan 1992; **1**(1):5–20, doi:10.1145/125489.125473. URL `http://doi.acm.org/10.1145/125489.125473`.
17. Offutt J, Abdurazik A. Generating tests from UML specifications. *UML'99*, Lecture Notes in Computer Science, Springer, 1999; 416–429.
18. Gnesi S, Latella D, Massink M. Formal test-case generation for UML statecharts. *9th IEEE International Conference on Engineering of Complex Computer Systems*, IEEE, 2004; 75–84.
19. Seifert D, Helke S, Santen T. Test case generation for UML statecharts. *Ershov Memorial Conference*, *Lecture Notes in Computer Science*, vol. 2890, Springer, 2003; 462–468.
20. Fröhlich P, Link J. Automated test case generation from dynamic models. *14th European Conference on Object-Oriented Programming (ECOOP 2000)*, *Lecture Notes in Computer Science*, vol. 1850, Springer, 2000; 472–492.
21. Utting M, Legeard B. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers, 2007.
22. Budd TA, Gopal AS. Program testing by specification mutation. *Computer languages* 1985; **10**(1):63–73.
23. Stocks PA. Applying formal methods to software testing. PhD Thesis, Department of computer science, University of Queensland 1993.
24. Ammann PE, Black PE, Majurski W. Using model checking to generate tests from specifications. *2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, IEEE, 1998; 46–54.
25. Aichernig BK, Peischl B, Weiglhofer M, Wotawa F. Protocol conformance testing a SIP registrar: An industrial application of formal methods. *5th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, IEEE, 2007; 215–224.
26. Fabbri SCPF, Maldonado JC, Sugeta T, Masiero PC. Mutation testing applied to validate specifications based on Statecharts. *10th International Symposium on Software Reliability Engineering (ISSRE'99)*, IEEE Computer Society, 1999; 210–219.
27. Hierons RM, Merayo MG. Mutation testing from probabilistic finite state machines. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION '07)*, IEEE, 2007; 141–150.
28. Briand LC, Penta MD, Labiche Y. Assessing and improving state-based class testing: A series of experiments. *IEEE Transactions on Software Engineering* 2004; **30**(11):770–793.
29. Black PE, Okun V, Yesha Y. Mutation operators for specifications. *15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, IEEE Computer Science, 2000; 81–89.
30. Hessel A, Larsen KG, Mikucionis M, Nielsen B, Pettersson P, Skou A. Testing real-time systems using UPPAAL. *Formal Methods and Testing*, 2008; 77–117.
31. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 2011; **37**(5):649–678.
32. Aichernig BK, Jöbstl E. Towards symbolic model-based mutation testing: Pitfalls in expressing semantics as constraints. *Workshops Proc. of the 5th Int. Conf. on Software Testing, Verification and Validation (ICST 2012)*, IEEE, 2012; 752 – 757.
33. Aichernig BK, Jöbstl E. Efficient refinement checking for model-based mutation testing. *12th Int. Conf. on Quality Software (QSIC 2012)*, IEEE, 2012; 21–30.
34. Aichernig BK, Jöbstl E, Kegele M. Incremental refinement checking for test case generation. *Tests and Proofs*, *Lecture Notes in Computer Science*, vol. 7942, Veanes M, Vigan L (eds.). Springer, 2013; 1–9.

DOI: 10.1002/stvr