Abstract—This paper presents the techniques and results of a novel model-based test case generation approach that automatically derives test cases from UML state machines. Mutation testing is applied on the modeling level to generate test cases. We present the test case generation approach, discuss the tool chain, and present the properties of the generated test cases. The main contribution of this paper is an empirical study of a car alarm system where different strategies for killing mutants are compared. We present detailed figures on the effectiveness of the test case generation technique. Although UML serves as an input language, all techniques are grounded on solid foundations: we give UML state transition diagrams a formal semantics by mapping them to Back’s action systems.

Keywords—test case generation; mutation testing; ioco; action systems; Unified Modeling Language

I. INTRODUCTION

Today’s dependable computer-based infrastructures rapidly grow in complexity due to a continuous evolution towards very large, heterogeneous, highly dynamic and ubiquitous computer systems. This trend of a growing complexity is a serious challenge to the task of engineering trustworthy systems: the more complex a system is, the more difficult is the verification of the fulfillment of its dependability requirements. 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 can be applied. In this work we present a formal model-based testing approach for generating test cases from UML state machines. Two features make our testing formal: (1) we give UML a formal semantics, and (2) we rely on a sound testing theory including a precise notion of conformance.

In order to give UML a formal semantics, we have developed a mapping from UML state machines to labeled transition systems via an object-oriented action system formalism. By translating UML models to labeled transition systems (LTS) we gain access to the existing testing theories based on LTS: in order to support partial, non-deterministic models, we distinguish between controllable (input) and observable (output) labels, and use the conformance relation \textit{ioco} [2] as a basis for test case generation.

We follow a classical and a non-classical test selection strategy. The classical approach is implemented via a mapping from our LTS interpretation of action systems to the CADP\textsuperscript{i} tools. This gives us access to the TGV test case generator [3] as well as to model checkers and simplifiers of CADP. The latter we use for checking both, the models as well as the translation chain from UML.

The non-classical test generation technique emphasized in this paper is model-based mutation testing. It adds a fault-centered testing approach to the tool chain. Program mutation testing provides a method of assessing and improving a test suite by checking if its test cases can detect a number of injected faults in a program. The faults are introduced by syntactically changing the source code following patterns of typical programming errors [4], [5]. However, in our approach we apply model-based mutation testing. The idea is to mutate the UML models and generate those test cases that would kill a set of mutated models. The generated tests are then executed on the system under test (SUT) and will detect if a mutated UML state machine has been implemented. It is a complementary testing approach, well-suited for dependability analysis, since its coverage is measured in terms of faults.

The mapping from UML and the general idea behind our test case generator have been described in detail elsewhere [1], [6], [7], [8]. In this paper, we focus on the characterization and evaluation of different test selection strategies. For illustration purposes, we have chosen a small demonstrator provided by our industrial partner Ford as a case study. Although the example is small, the tools are designed to cope with more complex control problems including interlocking of railway systems.

The rest of the paper is structured as follows. Section II presents the model of our running example, discusses the mutations, and sketches our mapping from UML to LTS. The main contributions of this paper can be found in Sections III and IV. Section III introduces a formal characterization of our resulting test cases and an algorithm for the generation

\textsuperscript{1}http://www.inrialpes.fr/vasy/cadp/
of adaptive test cases. Section IV presents different test selection strategies and the empirical results gained when applying them to our case study. Before concluding the paper in Section VI, we discuss related research in Section V.

II. A UML MODEL

In order to demonstrate the basic concepts of our test case generation approach, we use a simplified version of a car alarm system (CAS). The example is taken from Ford’s automotive demonstrator within the MOGENTES project. The following requirements were specified and served as the basis for our 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 unauthorized 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.

A. Testing Interface

As shown in Figure 1, the UML model resulting from the above stated requirements comprises four classes and four signals. The class `AlarmSystem` is labeled as system under test (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 labeled as being part of the environment. In this way, the class diagram in Figure 1 specifies the observations we can make (all calls to methods being part of the environment) and the stimuli the SUT can take (all signals). Hence, the diagram specifies our testing interface.

B. State Machine

Figure 2 shows our 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 modeled 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 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) we decided to treat the underspecification in the requirements in a way 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.

C. Mutations

Since we want to create test cases that cover particular fault models, we deliberately inject ‘bugs’ into the specification and then create test cases that identify the differences. Therefore, we use various mutation operators. For example, one mutation operator sets guards of transitions to false. Others remove entry actions, signal triggers, or change signal events.

For the CAS specification depicted in Figure 2, we obtain 76 mutated UML state machines. 19 mutants with a transition guard set to false, 6 mutants with a missing entry action, 12 mutants with missing signal triggers, 3 with missing time triggers, and 36 with changed signal events.

We create first-order mutants, i.e., each mutated state machine covers only one particular mutation (one mutation operation in a particular place). Mutation testing relies on two assumptions: (1) competent engineers write almost correct code, i.e., faults are typically “one-liners” and (2) this may lead to a transition transforming into a self-loop as the model will ‘swallow’ the trigger event.
there exists a coupling effect so that complex errors will be found by test cases that can detect simple errors.

Note that our approach also addresses the problem of equivalent mutants up to a certain search depth. We discover equivalent mutants during test case generation and skip them, since they show no deviating behavior we can test for.

D. From UML to LTS via Action Systems

Since our algorithms for test case generation work on the level of Labeled Transition Systems (LTS), the UML state machine model defined by the user has to be converted into an LTS. Basically, this conversion comprises the following steps:

1) The given UML model is transformed into a labeled and object-oriented intermediate representation that is based on Back’s action system [9] formalism. During this step we set the exact UML semantics, e.g., the treatment of UML events and that of parallel regions. As a result, we derive an executable model of the system.

2) The object-oriented intermediate representation is simplified and a non-object-oriented (but still labeled) version of the action system is generated.

3) The action system derived in the second step is explored. This process yields the LTS of the UML model: Overly simplified, the execution of one labeled action within the action system adds one transition to the LTS.

For further information about this conversion, we refer to our descriptions in [1].

We have developed tools that automate this process. Applying these tools to our model of the car alarm system (see Figure 2) yields the LTS shown in Figure 3. All labels in our system are prefixed by either obs (observable) or ctr (controllable). As explained in Section II-A, 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 III-A).

III. TEST CASE GENERATION

In this section, we explain our test case generation technique. First, we introduce the conformance relation that allows us to compare original and mutated specs. Then, we briefly explain our conformance checker. Next, we discuss the properties of the generated test cases and present an algorithm for test case selection.

A. Input-Output Conformance

Conformance relations are used to determine if a system under test (SUT) behaves correctly regarding a given specification. In our case, the SUT is represented by the mutated testing model while the specification is formed by the non-mutated testing model. In order to decide conformance, some testing hypotheses have to be stated [10]. One is that the implementation can be represented with the same formalism as the specification. In our application this is always the case since we decide the conformance between two specifications (an original and a mutated version). Since we are interested in input-output testing with partial system models we apply the ioco relation [2].

The trace semantics of an action system, see Section II-D and the work in [7], yields a Labeled Transition System (LTS) M. Thereby, a labeled transition system is defined as tuple \( (S, L, T, s_0) \) 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 \( L_I \cap L_U = \emptyset \)
- \( T \subseteq S \times (L \cup \{ \tau \}) \times S \) is the transition relation, and
- \( s_0 \in S \) is the initial state.

The special label \( \tau \) denotes an internal action. We use the following notation for a given LTS, where \( s, s', s_i \in S, S' \subseteq S \)
$S_i, a_i \in L, \sigma \in L^*$.

$s \xrightarrow{a} s' =_d (s, a, s') \in T$

$s \xrightarrow{a} =_d \exists s' \bullet (s, a, s') \in T$

$s \xrightarrow{a} =_d \exists s' \bullet (s, a, s') \in T$

$s \xrightarrow{\tau} s' =_d \exists s' \vee \exists s_0 \ldots s_n \bullet$

$s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_{n-1} \xrightarrow{\tau} s_n = s' $

$s \xrightarrow{a} s' =_d \exists s_1, s_2 \bullet s \xrightarrow{\tau} s_1 \xrightarrow{a} s_2 \xrightarrow{\tau} s' $  

$s \xrightarrow{\tau} s' =_d \exists s_0, \ldots, s_n \bullet s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_n = s' $  

with $\sigma = a_1 \ldots a_n$

$s \xrightarrow{a} =_d \exists s' \bullet s \xrightarrow{\tau} s' $  

$s \text{ after } \sigma =_d \{ s' \mid s \xrightarrow{\tau} s' \}$

The relation after\textsuperscript{4} determines the set of states reachable after a trace $\sigma$ starting from an initial state. Moreover, an LTS $M$ has finite behavior if all traces have finite length. It is deterministic if $\forall \sigma \in L^* \bullet |s_0 \text{ after } \sigma| \leq 1$ holds.

For the ioco relation, SUTs are considered to be weak input-enabled, i.e., all inputs (possibly preceded by $\tau$ transitions) are enabled in all states: $\forall a \in L_I, \forall s \in S \bullet s \xrightarrow{a}$. 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: $\forall a \in L_U \cup \{\tau\} \bullet s \xrightarrow{a}$. For observing quiescence, the transition relation $T$ is extended by adding self-loops with the special label $\delta$ at quiescent states: $T_d =_d T \cup \{(s, \delta, s) \mid s \in S \land \delta(s)\}$. Let $M_d$ be the LTS over the alphabet $L \cup \{\tau, \delta\}$ resulting from adding $\delta$ self-loops to an LTS $M$. Then the deterministic version of $M_d$ is called suspension automaton $\Gamma$. The set of suspension traces is:

$$Straces(M_d) =_d \{ \sigma \in (L \cup \delta)^* \mid s_0 \xrightarrow{\tau} \}$$

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:

$$out(s) =_d \{ a \in L_U \mid s \xrightarrow{a} \} \cup \{ \delta \land \delta(s) \}$$

$$out(S') =_d \bigcup_{s \in S'} out(s)$$

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:

$$i \xrightarrow{ioco} s \text{ if } \forall \sigma \in Straces(s) \bullet out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$$

\textsuperscript{4}Not to be confused with the timeout observation $after (t)$ in Figure 3.

B. Conformance Checking

We have implemented a tool for checking the input-output conformance of two system models named Ulysses. Figure 4 depicts the computation steps of our tool. Ulysses expects two labeled action systems as input: (1) a system specification $AS$ and (2) a mutated version of the same specification $AS^M$. Section II-D has already dealt with the transformation from a UML model to an LTS. The last step of this conversion is actually performed by the Ulysses tool by exploring a given labeled action system which yields its LTS semantics, see [7] for further details. Then the LTS is enriched by quiescence and subsequently converted into a deterministic automaton. By executing these steps, which are depicted in the first box of Figure 4, we have gained a so-called suspension automaton.

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), which generates a product graph from which we extract controllable test cases (last box in the Ulysses process).

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, see [7].

Consider the example in Figure 5. The first two LTSs show the partial suspension automata of the specification and a mutant of the Car Alarm System. In the mutant, the controllable action Lock is a self-loop. The mutant is not ioco to the specification because the subset inclusion of the observations after the trace $\langle Lock, Close \rangle$ does not hold, i.e., $out(Mutant, after (Lock, Close)) = \{\delta\} \not\subseteq out(Spec, after (Lock, Close)) = \{after(20)\}$. The product LTS (depicted on the right-hand side) may contain states with pass self-loops. They denote that the specified behavior is left after an unspecified observable event or is aborted after a not implemented observable event. Furthermore, fail states, i.e., states with fail self-loops, denote that an unspecified observable event has occurred. In this case, we are able to derive a controllable test case that is able to detect this unspecified behavior (cf. LTS on the right-hand side).

C. Test Case Selection and Coverage

In the following we first show the required properties of our selected test cases and then present a selection algorithm for adaptive test cases.
Given a product LTS \((S^P, L^P, T^P, s_0^P)\) being the result of an ioco check and a set \(\text{Fail} \subseteq S^P\) denoting the set of fail states. We define an unsafe state as a state after which we may fail.

\[
\text{Unsafe} =_a \{ s \in S^P \mid s \xrightarrow{a} f \land f \in \text{Fail} \land a \in L_U \}
\]

Unsafe states play the central role in our test case generation strategy, since they are the test goal a test case should cover. Below, we present the general properties of a generated test case LTS \(TC = (S^TC, L^TC, T^TC, s_0^TC)\) that is selected from a product.

\[
S^TC \setminus \{\text{pass, inconc}\} \subseteq S^P \land L^TC \subseteq L^P \land s_0^TC = s_0^P
\]

(Sink states)

\[
s \xrightarrow{a} \iff s \in \{\text{pass, inconc}\}
\]

(Passing)

\[
s = \text{pass} \iff \exists u \in \text{Unsafe} \bullet u \xrightarrow{a} s \land s \notin \text{Fail} \land a \in L_U
\]

(Contrallability)

\[
\exists! u \in S^TC \bullet (u \in \text{Unsafe}) \land u \xrightarrow{a} \text{pass}
\]

(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.

\[
s \xrightarrow{TC} \text{inconc} \iff s \xrightarrow{TC} u \land u \in \text{Unsafe} \land\exists s' \bullet ((s^TC, a, s') \in T^P \land a \in L_U \land a \notin \text{hd}(\sigma))
\]

(Inconclusive linear)

Such a test case contains exactly one path to the unsafe state (linear test case). Since a model’s behavior 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 \(\text{hd}\) returns the first element in the trace \(\sigma\).

A test case is a sub-transition system of the calculated product extended with two additional verdict states \(\text{pass}\) and \(\text{inconc}\) (contrallability). In the test case, a sink state is a verdict state \(\text{pass}\) or \(\text{inconc}\). Note, fail verdicts are implicit and not included (sink states). The pass verdict is characterized by successfully passing an unsafe state (passing). A test case does not contain choices over controllables, see also [2] (contrallability). Reaching an unsafe state is the test goal of our mutation testing strategy. Hence, per test case exactly one unsafe state precedes a pass verdict state (test goal unique). 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 our 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, we discuss the properties of the two kinds.

**Linear Test Cases:** are necessary if the target test harness does not support branching behavior. The following two additional properties characterize linear test cases:

\[
\exists! \sigma \bullet s_0^TC \xrightarrow{\sigma} u \land u \in \text{Unsafe} \land\exists s' \bullet ((s^TC, a, s') \in T^P \land a \in L_U \land a \notin \text{hd}(\sigma))
\]

(Inconclusive linear)

Note the difference to 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 we are unable to reach the unsafe state.

\[
TC_u =_a \text{getTC}(s_0, \{u\}) \cup \{(u, a, \text{pass}) \mid \exists s \bullet u \xrightarrow{a} s \land a \in L_U \land s \notin \text{Fail}\}
\]

(Adaptive test case)

Algorithm 1 describes the selection of adaptive test cases, which is represented by the extraction and contrallability box in Figure 4. The input of this algorithm is the product LTS and an unsafe state \(u\). The product LTS is
constructed on-the-fly as described in Section III-B either until a difference regarding ioco is found or a defined search depth is reached. In the following we denote variables with capital letters. We obtain an adaptive test case TC\(_i\) constructed on-the-fly as described in Section III-B either for the shortest trace from the initial state to an unsafe state. The recursive algorithm terminates when all branching outputs have been processed. Finally, the graph obtained from getTC is extended with the transitions to the pass state, see Definition (adaptive test case).

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

1: if \( s \in \text{Goal} \) then 
2: \hspace{1em} return \( \emptyset \)
3: else 
4: \hspace{1em} \( \sigma := \text{shortestTrace}(s, \text{Goal}) \)
5: \hspace{1em} if \( \sigma = \emptyset \) then 
6: \hspace{1em} \hspace{1em} return \( \{(s, a, \text{inconc}) \mid \exists s' : s \xrightarrow{a} s' \land a \in L_U \land s' \notin \text{Fail}\} \)
7: \hspace{1em} else 
8: \hspace{1em} \hspace{1em} \( E := \text{edges}(\sigma) \)
9: \hspace{1em} \hspace{1em} \( \text{Goal} := \text{Goal} \cup \text{states}(E) \)
10: \hspace{1em} \hspace{1em} \text{getTC}(s_2, \text{Goal}) \)
11: \hspace{1em} end if
12: end if

In Line 1 it is checked if the current state is already a goal state. If it is a goal state the empty set is returned. In Line 4 the shortest trace between a state \( s \) and one state in the set of goal states is determined. Note that there can exist several shortest traces of same length. In this case, we nondeterministically choose one. If there exists no such trace then all observations in state \( s \) terminate in an unsafe state and recursively extending this trace for all branching output transitions, the obtained test case complies with the controllability requirement (controllability).

In Line 9, the set of states in \( E \) obtained by the function \( \text{states} \) 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. The transitions of a trace are determined by the function \( \text{edges} : (L \cup \delta)^* \rightarrow \mathcal{P}(S \times (L \cup \delta) \times S) \).

In Line 10, the set of tests generated by the approach \( A_2 \) have been generated via straightforward path search the approaches \( A_3 \sim A_6 \) employ the test case algorithm 1. In \( A_7 \) we generate test cases randomly and due to algorithm 1. Table I highlights the main differences between the approaches regarding generated test cases.

### Approach A1:
Within the first approach, the product graph of the ioco check is first transformed into a tree structure with a maximum depth of 10 and a two-times maximum visit per state per trace. After tree unfolding the graph, we generate linear test cases for all paths that lead to an unsafe state in the tree. As can be seen in the table, 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, we could exclude another 562 tests that are a complete subsequence of one of the other remaining test cases. Hence, about 3500 tests remain. Due to the vast amount of test cases generated by \( A_1 \), we refrained from executing the tests on the CAS implementations.

### Approach A2:
The second test case generation strategy directly works on the product graph and extracts only one arbitrary linear test per unsafe state. Notice that one mutant may still involve more than one unsafe state in the product graph. This time, we could identify 58\% of duplicate test cases and another 22 were covered as a subsequence in other test cases. It has to be said that we allowed for tests with a maximum length of 14 in this approach, thus generation times are not directly comparable between \( A_1 \) and \( A_2 \). We verified that all unique test cases with a length of up to 10 that were produced by this approach were included in the set of tests generated by the first approach.

### Approach A3:
The third approach is similar to the second one, except that the depth is theoretically unbounded since the test case generator applies algorithm 1. Therefore, the generation time is not comparable with approaches \( A_1 \) and \( A_2 \) (but with \( A_4 \) to \( A_7 \)).

### IV. Experimental Results

In the preceding sections, we have discussed the theoretical basis of our test case generation approaches. This section goes one step further as we present the empirical results gained when applying eight different test case generation approaches to the car alarm system. We start our discussion with a summary of the different test case generation approaches before evaluating their fault detection capabilities.

#### A. Test Case Generation Results

We compare eight different test case generation approaches called \( A_1 \) to \( A_8 \). Generally speaking, \( A_1 \) to \( A_6 \) are fault-based approaches, \( A_7 \) is based on random testing, and approach \( A_8 \) uses test cases that were manually designed. All fault-based approaches use a set of 76 faulty specifications (cf. Section II-C) as input to test case generation. Note that because the car alarm system is deterministic, all test cases are linear. While the test cases in approach \( A_1 \) and \( A_2 \) have been generated via straightforward path search the approaches \( A_3 \sim A_6 \) employ the test case algorithm 1. In \( A_7 \) we generate test cases randomly and due to algorithm 1. Table I highlights the main differences between the approaches regarding generated test cases.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Test Case Generation</th>
<th>Fault Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Greedy</td>
<td>100%</td>
</tr>
<tr>
<td>A2</td>
<td>Greedy Linear</td>
<td>75%</td>
</tr>
<tr>
<td>A3</td>
<td>Random</td>
<td>58%</td>
</tr>
<tr>
<td>A4</td>
<td>Random Linear</td>
<td>22%</td>
</tr>
<tr>
<td>A5</td>
<td>Random Linear</td>
<td>22%</td>
</tr>
<tr>
<td>A6</td>
<td>Random Linear</td>
<td>22%</td>
</tr>
<tr>
<td>A7</td>
<td>Random</td>
<td>562 tests</td>
</tr>
<tr>
<td>A8</td>
<td>Manual</td>
<td>0 tests</td>
</tr>
</tbody>
</table>
Approach A4: The fourth approach builds on A3 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 we generate a test purposes $tp$ 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[3], 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 $u$ we determine the set of states $L2U$ which lead to $u$, i.e. $L2U = \{ s \in S \mid \exists \sigma \bullet s \xrightarrow{\sigma} u \}$. The set of pass states associated with $u$ are $Pass = \{ s \in S \mid \exists a \in L_U \bullet (u, a, s) \in T \land s \not\in Fail \}$. The resulting test purpose $tp_u$ has the transition relation $T_u = \{ (s_1, a, s_2) \in T \mid s_1, s_2 \in L2U \land (s_1 = u \land s_2 \in Pass) \} \cup \{ (s, *, s) \mid s \in Pass \}$. The remaining behavior after reaching a state $s \in Pass$ is not of interest, denoted by the $*$ label allowing any transitions to occur thereafter.

This approach yields one test case per unsafe state. Note however, that generated test cases still can be included in other test cases. This depends on the order of processed unsafe states. If we would start with deep unsafe states and proceed to shallow ones it is theoretically possible that no generated test case is included in another one.

While the total number of generated test cases decreases to 129 (from 504), the time used to generate them increases to 70 minutes (from 23). This is because we compute the check for every unsafe state and in the worst case for all test cases generated so far. In effect, A4 checks whether an existing test case already covers an unsafe state in the $ioco$ product before creating a new test case. If an unsafe state (there may be several per mutated specification) is not covered, a new test case is emitted.

Approach A5: Approach A5 also avoids creating duplicate test cases. However, A5 further tries to minimize the size of the generated test suite. Before creating test cases for a mutated specification, A5 first checks whether any of the previously created test cases is able to kill the mutated specification. Therefor, the synchronous parallel execution $\parallel$ of the test case $tc$ with the mutated specification under test $m$ is applied. The following rules define the operational semantics for killing a mutated specification.

$$\begin{align*}
    tc \xrightarrow{\sigma} tc' \quad m \xrightarrow{\sigma} m' & \quad (1) \quad \frac{a \in L_U \quad tc \rightarrow s \quad m \rightarrow m'}{tc \parallel m \rightarrow tc' \parallel m'} \\
    m \xrightarrow{\sigma} m' & \quad (2) \\
    a \in L_U \quad tc \rightarrow s \quad m \rightarrow m' & \quad (3)
\end{align*}$$

Rule (1) describes the case where $tc$ and $m$ synchronize 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 realized 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 a mutant is killed when a fail state can be reached, i.e.

$$\exists \sigma \in (L \cup \delta)^* \bullet tc \parallel m \rightarrow fail \parallel m'$$

The check is done without calculating the full $ioco$ product between the specification and the mutant, which is the first difference to A4. A4 always calculates the full $ioco$ product. 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. Notice that this is the second difference to A4 as A4 will only skip test case generation for covered unsafe states, while A5 does never generate any additional test case for a killed mutant. Due to this minimization, approach A5 is sensitive to the ordering of mutants. Notice that A5 starts with an empty test suite.

Approach A6: Approach A6 is like A5 but instead of starting with an empty test suite, A6 uses one randomly generated test case to start with. In effect, A6 is a combination of random (A7) and fault based testing (A5). Within Table I, 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.

Approach A7: This approach uses a random selection strategy in order to generate test cases. Put differently, A7 is not a fault-based test case generation approach and included for evaluation purposes only.
Approach A8: Finally, we compare our test case generation approach based on UML with manually created test cases. More precisely, we generated 9 different test purposes by hand and let TGV [3] create test cases. 3 out of the 9 test cases check for observable timeouts (time-triggered transitions: 20, 30, 300 sec. delay). 4 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 and 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. During the creation of the test purposes, we relied on a printout of the UML state machine.

B. Test Case Execution Results

We applied classical mutation analysis in order to evaluate the effectiveness of our different test suites. For this purpose, we have implemented the CAS in Java based on the state machine of Figure 2. In order to derive a set of faulty implementations, we used the μJava [11] tool: in total, μJava gave us 72 mutated implementations. After careful 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 equivalent to other mutants - forming 26 equivalent pairs. Hence, a sum of 38 (72-8 - 26) different faulty implementations of the CAS remain. Further details are shown in Table II. 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 ones and handle the equally named external events while SetState and the constructor (Constr) are internal methods. From the table, one can observe that the mutation on internal methods has a strong effect on the external behavior since there are no equivalent mutants for these methods.

In the following, we use the 38 unique faulty CAS implementations to evaluate the effectiveness of our generated test cases. Of course, all tests were validated on the non-mutated CAS implementation.

Table III gives an overview of the number of survived faulty implementations for each test case generation approach. As can be seen from the table, the approaches A3, A4, and A6 were able to reveal all faults. The table also proves that the minimization algorithm applied in approach A5 reduces the fault-detection capabilities. Despite random testing (A7) did not find all faulty implementations, it proved quite effective in our setting. It has to be noted, however, that we allowed for an appropriate depth of the random test cases. Summing up, A5 and A7 still have a fault detection rate of 97%. The reason for the bad performance of A2 is the fact that the two surviving faulty implementations need tests with a depth of more than 14 interactions to be revealed and A2’s tests are restricted to a depth ≤ 14.

The last column shows the results of running the manually designed tests (A8). Overall, 25 mutants were killed which results in a detection rate 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 behavior 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, we have measured the coverage on the implementation in terms of basic block coverage. The approaches A2 - A7 achieve a coverage of 99%, whereas A8 (TGV) only results in a basic block coverage of 88%. By comparing the last two rows of Table III, it becomes obvious that a high code coverage does not automatically guarantee high fault detection rates, which we aim for.

In summing up, the results show that our approaches are powerful in covering the implementation’s source code and more importantly in detecting faults. However, the depth of our conformance analysis is critical as too less 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 approach were by far too many: for the given
model mutations, one path per mutation is sufficient to detect all faults, provided this path is long enough. Finally, the combined approach (A6) proved to be a nice trade-off between generation time and effectiveness.

Beside using UML models we directly modeled the CAS in OOAS language. The transformation from UML to OOAS brings quite a big overhead: while the hand-written OOAS model contains 105 lines and 11 actions, the transformed UML model results in 750 lines of code and 55 actions. This is mainly because UML is intended to specify systems on the implementation level rather than on an abstract requirement level. For instance, the whole event signaling mechanism of UML has to be transformed to OOAS constructs.

For a comparison with the UML-based approach we implemented a mutation tool which created 442 mutated OOAS specifications. Table IV shows the number of test cases with according generation times for approaches A3-A5. Each test suite of the three approaches was able to detect all faulty implementations. The results show that our selection strategies are able to efficiently minimize test suites for a large number of mutated specifications.

V. RELATED RESEARCH

There exist already various research papers on test generation from UML state charts. Gnesi et al. [12] define the semantics of a subset of UML state charts in the form of transition systems with 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. [13] define so-called Compact Semantic Automaton (CSA) in order to efficiently represent the semantics of a UML state chart. From CSAs, they derive traces of observable events which serve as input for the test case generation algorithm defined by Tretmans [2]. Fröhlich et al. [14] systematically transform use cases into UML state charts and generate test suites with a given coverage level by applying AI planning methods.

Model-based mutation testing was first used for predicate-calculus specifications [15]. Later, Stocks applied mutation testing to formal Z specifications [16] without automation. Due to model checking techniques, full automation became feasible. By checking temporal formulae that state equivalence between original and mutated models, counterexamples are generated, which serve as test cases [17]. In contrast to this state-based equivalence test, we check 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 [18]. To our knowledge, we are the first who 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 [19]. It seems that we are the first who actually generate test cases from mutated UML state machines.

The work in [20] deals with mutation-based test case generation for probabilistic finite state machines. The authors present mutation operators and describe how to create input sequences to kill a given mutated state machine. In [21] state-based testing of classes is investigated. The authors present results for testing state charts and evaluate their approaches on a set of mutated implementations.

Regarding time, we do not fully support the interleaving semantics of parallel running timers like UPPAAL [22].

VI. CONCLUSION

We have presented our UML-based test case generation approach in a complete and concise manner. Other novelties covered in this paper are the formalization of our test cases’ properties and the adaptive test case selection algorithm.

The main contribution, however, is the characterization and comparison of six different fault-based test case selection strategies. All of the presented strategies have been implemented in our tool chain. When fed with an UML model as input, it generates a test suite of possibly adaptive, ioco-conforming test cases.

We have shown the effectiveness of our approaches in comparison to random testing and manual test case generation. The experiments indicate that the most promising approach is a clever combination of random exploration and conformance checking (Approach A6). The former is fast but incomplete with respect to covering the model mutations. The latter is complete, but expensive.

The car alarm system is small but non-trivial: it involves hierarchical states, entry-exit actions with methods defined in class diagrams. Currently, we are working on larger case studies and the refinement of our tools. Furthermore, we investigate approaches to minimize the needed mutations.

ACKNOWLEDGEMENT

Research herein was funded by the EU FP7 project ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems (MOGENTES). We want to thank our colleagues Rupert Schlick and Wolfgang Herzner from AIT (Austrian Institute of Technology) for the discussion on the paper. They also prepared the initial UML model, the UML mutation operators, and – in collaboration with the authors – defined the UML to OOAS mapping. Johannes Reiter (AIT) helped in developing the UML to OOAS conversion tool. The requirements for the CAS example were provided by Ford.
REFERENCES


