UML in Action: A Two-Layered Interpretation for Testing[*]

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl
Institute for Software Technology
Graz University of Technology
Austria
{aichernig, hbrandl, ejoebstl}@ist.tugraz.at

Willibald Krenn[†]
Maxeler Technologies, Ltd
1 Down Place
London W6 9JH, UK.
willibald.krenn@gmx.at

Abstract

This paper presents a novel model-based test case generation approach that automatically derives test cases from UML state machines. UML is given a two-layered formal semantics by (1) mapping UML class diagrams and state charts to Back's Action Systems, (2) by interpreting these action systems as labeled transition systems. The first semantics provides a formal framework to capture the object-oriented machinery: classes, objects, inheritance, transitions, time-outs, signals, nested and parallel regions. The second mapping represents the tester's view on the interface in terms of input and output actions. Tretman's input-output conformance relation (ioco) forms the basis of our fault models. Mutation analysis on the models is used to generate test cases. A car alarm system serves as a running example.

1 Introduction

Today, the transportation industry is challenged by the exponential growth in embedded systems complexity, while at the same time meeting tight cost constraints and facing the need to further reduce time to market. Taking into account the critical role of Validation & Verification, this work addresses the need to advance the current industrial testing techniques. We do this by exploiting the growing modeling expertise in industry using UML and present a novel model-based test case generation approach that automatically derives test cases from UML state machines. The novelty of our work lies in the combination of existing formal techniques integrated into a tool set targeting industrial acceptance.

In order to give UML a formal semantics, we have developed a new and non-trivial mapping from UML state machines to an object-oriented version of action systems [9]. We chose action systems, because they are well suited to express both, control problems, which are common in embedded systems, and the kind of concurrency found in UML state machine models. Furthermore, the object-oriented extension of action systems facilitates the mapping of the test interfaces defined in class diagrams.

Action systems are state-based and give UML a weakest-precondition semantics. However, for black-box testing we are interested in the external communication events only. Therefore, we label all actions and interpret them as a labeled transition system (LTS). This semantic abstraction gives us 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) action labels, and use the input-output conformance relation *ioco* [25] as a basis for test case generation. With this two-layered semantics, we are able to compare models from a developer's point of view (state-based refinement) and from a tester's point of view (input-output conformance). These solid foundations grounded in both worlds are essential for developing trustworthy tools.

With respect to 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[1] tools. This gives us access to the TGV test case generator [18] 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.

Our non-classical test generation technique emphasized in this paper is *model-based mutation testing*. It enriches the tool chain by considering fault-centered aspects. Originally, mutation testing is a way of assessing and improving a test suite by checking if its test cases can detect a number of faults injected into a program. The faults are introduced by syntactical changes of the source code following patterns of typical programming errors [17, 13]. However, we apply model-based mutation testing in which the basic idea is to mutate the UML models and generate those test cases that kill the set of mutated models. The generated tests are then executed on the system under test (SUT) and 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 contributions of this work can be summarized as follows: (1) a new formal semantics of UML state charts in terms of object-oriented action systems, (2) a new mutation testing approach for UML based on labeled transition systems, (3) full tool support for the semantic mappings and the test case generation, (4) empirical evidence that the method works. This is the first time that we present an overview of our techniques leading to a formal testing approach for UML.

The rest of this paper is structured as follows. In Section 2

[*]Research herein was funded by the EU FP7 project ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems (MOGENTES).

[†]W. Krenn carried out this work when still working at the Graz University of Technology.

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

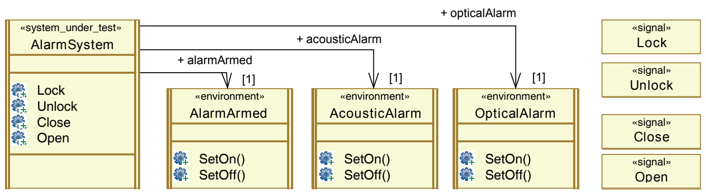

Figure 1: Car Alarm System - Testing Interface.

we introduce our running example, the kind of UML models we are targeting, and the mutation thereof. In Section 3 we explain our mapping from UML to object-oriented action systems (OOAS) and how we interpret the actions as labeled transition systems. Next, in Section 4 we present our notion of conformance and in Section 5 we give an overview of our conformance checker tool. We discuss the empirical results of our case study in Section 6. Finally, we present related work in Section 7 and conclude in Section 8.

2 A UML Model

For demonstrating the main concepts of our approach, we use a simplified car alarm system. The example is taken from Ford's automotive demonstrator within the MOGENTES project. The UML testing model for the car alarm system (CAS) was created from the following list of requirements:

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 CAS can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

When trying to construct an animated model based on textual requirements, often conflicts or underspecified situations become apparent. One might think that the simplistic car alarm system is sufficiently described by these three textual requirements – the contrary is the case. What is left unspecified is the case of what happens when an alarm is ended by the five minutes timeout: does the system go back to armed directly, or does it need to wait for all doors to be closed again before returning to armed? For our model, we chose the latter option.

Testing Interface. The UML model of the CAS comprises four classes and four signals (cf. Figure 1). The class *AlarmSystem* is marked 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 marked as being part of the environment.

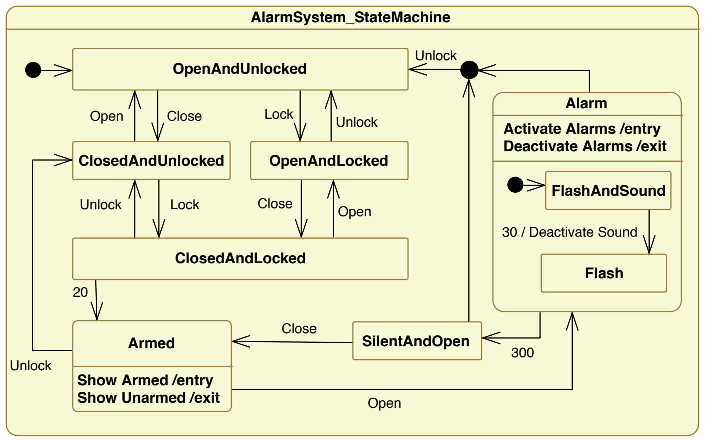

Figure 2: Car Alarm System - State Machine.

Note that the class diagram in Figure 1 specifies the observations (all calls to methods being part of the environment) we can make and the stimuli the SUT can take (all signals). In effect, this diagram specifies our testing interface.

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 modeled by corresponding signals *Close*, *Open*, *Lock*, and *Unlock*. As specified in the first requirement, 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. Similar, 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, turning off the acoustic alarm after 30 seconds, as specified in the second requirement, is reflected in the time-triggered transition leading to the *Flash* sub-state of the *Alarm* state.

Mutations. As we want to create test cases that cover particular fault models, we need to deliberately introduce 'bugs' in the specification model. In order to do that, we rely on different mutation operators. As an example, one mutation operator sets guards of transitions to false[2], while others remove entry actions, signal triggers, or change signal events. Applying these operators to our CAS specification model yields 76 mutants: 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.

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

One particular mutant that we will also refer to in one of the following sections, lacks the transition from *OpenAndUnlocked* to *OpenAndLocked*. This means that the system simply ignores the *Lock* event when being in the *OpenAndUnlocked* state instead of proceeding to *OpenAndLocked*.

Each mutant covers only one particular mutation (one mutation operation in a particular place): Mutation testing is based on the assumptions that (A) competent engineers write almost correct code, i.e., faults are typically "one-liners" and that (B) there exists a coupling effect so that complex errors will be found by test cases that can detect simple errors. Note that we discover equivalent mutants during test case generation and skip them, since they show no deviating behavior that we could test for.

3 UML Semantics

Object-oriented action systems [9] (OOAS) are an extension to the action system formalism initially proposed by Back et al. in [5, 6]. Our UML semantics in terms of labelled OOAS is the first to combine OOAS (with custom extensions), prioritizing composition [23], complex data types, and an event-trace semantics of action systems. We start with the introduction of classical (non-object-oriented) action systems.

3.1 Action Systems

We represent an action system AS comprising n methods, m named actions A, d action calls $C_i \in A$ composed via one of the operators $\mathbf{op} =_{df} \;\Box \;|\; ; \;|\; //$, and a set M_I of imported methods syntactically as follows:

$$\begin{array}{lll} AS =_{df} & |[& \mathbf{var} \quad V : T = I \\ & & \mathbf{methods} \\ & & M_1; \ldots ; M_n \\ & & \mathbf{actions} \\ & & A_1; \ldots ; A_m \\ & & \mathbf{do}\ C_1 \;\texttt{op}\; C_2 \ldots \texttt{op}\; C_d \;\mathbf{od} \\ &]| : M_I & \end{array} \tag{1}$$

Note that methods have a name, a body and may have a return value. Named actions are similar to methods but have no return value. The box-operator (" $\Box$ ") stands for non-deterministic, demonic choice. Demonic choice of actions means that when an aborting action is enabled, this action is chosen. The operator ";" denotes sequential composition, and the "//" operator stands for prioritizing composition. A prioritizing composition $A//B$ means that if A is enabled it will be executed, otherwise action B is executed. This operator can be rewritten into a non-deterministic choice, namely $A \;\Box\; \neg g.A \rightarrow B$, where $g.A$ denotes the enabledness guard of action A.

In the remainder of this paper, we assume that named actions may only be called from within the **do od**-block (that is, not from within named actions or methods), and that method calls must not be recursively nested. We also demand that each named action has the form of a guarded command. Relying on these assumptions, we are allowed to rewrite the action system into a more classical form, where only the actions within the **do od**-block are left:

$$AS \quad =_{df} \quad |[\; \mathbf{var} \quad V : T = I \quad \mathbf{do}\ A\ \mathbf{od}\;]| : Z$$

Within this representation, V is a vector of variables of types T, initialized with values I, and actions A_i ($1 \leq i \leq m$) have been subsumed under action A. Finally, after "inlining" all imported methods $\in M_I$, Z denotes the set of imported variables of the environment that was accessed by the imported methods.

After eliminating all method calls, the action system consists of basic actions only and all actions are part of the **do od**-block, also known as Dijkstra's guarded iteration statement [14]. The guarded iteration statement can be thought of as being a loop that selects one enabled action A_i for execution in each iteration. In case there is no action enabled, execution of the action system ceases as execution of the loop terminates.

Like on actions, prioritizing composition is associative on action systems. However, it does not in general distribute over parallel composition to the right when used on action systems. This is due to local variables that would be duplicated.

3.2 Object Orientation

The work of Bonsangue et al. [9] forms the basis for our object-oriented formal semantics: in particular we share the transformation step from object-oriented action systems to action systems. We differ in the notion of named actions and procedures and we add the ability to prioritize objects of a particular class with respect to objects of another class. Within our methodology, we use a very simple form of inheritance: A class C^2 is a valid subclass of C^1 if and only if the (syntactic) superposition (cf. [7]) refinement holds between the classes. Roughly speaking this means that C^2 may introduce additional variables and actions. However, none of the additional actions may have any effect on the variables of C^1. It must be guaranteed that when only considering the new actions and the initial state the system terminates, and the exit condition of C^2 must imply the exit condition of C^1. The subclass C^2 may override (refine) actions of C^1 in a way that the guard is strengthened and values to the additional variables are assigned.

Like most object-oriented programming languages, objects are constructed at runtime from classes with the help of a constructor statement $o := new(C)$, where o represents the instance (object) and C stands for some class. A class C is a named type and can be represented as a tuple $C =_{df} (C_n, C_b)$ where $C_n \in \mathcal{CN}$ is a class name from the set of class names $\mathcal{CN}$ and C_b is the body of the type definition according to our action system syntax in (1). Note that methods are "public", as they can be called by any other method or action.

We restrict an object-oriented action system to a fixed set of classes $\mathcal{C} =_{df} \{C^1, \ldots, C^k\}$ and a fixed set of objects. Practically, this means that we allow object instantiation only during state variable initialization, which permits us a rather easy check of finiteness. When a class in an object-oriented action system is marked as *autocons*, one instance of the class will be created automatically at system start and is called a "root object".

We assume that all objects of one class have the same priority. Between objects of different classes, however, we allow ordering with the help of the prioritizing composition operator: we introduce a so-called system assembling block (*SAB*). The *SAB*, which is an extension to the work of [9], specifies the ordering of priorities between objects of different classes. We rely extensively on this feature in order to model, e.g., event broadcasting. The syntax of the system assembling block is defined by the following grammar:

$$SAB ::= C_n \ ((\ \Box \mid // \ )\ SAB)?$$

Note that the non-deterministic choice operator denotes parallel composition and the prioritizing composition operator expresses a prioritizing composition of objects. As an example, $C^1 // C^2$ means that only if there is no action enabled in any of the C^1 objects, actions of any of the C^2 objects will be looked at.

Hence, we define an object-oriented action system as a 3-tuple $(\mathcal{C}, \mathcal{R}, SAB)$, where $\mathcal{C}$ is a fixed set of classes $\{C^1, \dots C^k\}$, $\mathcal{R} \subseteq \mathcal{C}$ is a set of classes that need to be instantiated once at system start, and SAB is the system assembling block. Within the system assembling block, each class name $C^i_n \in \mathcal{C}$ must be listed once, and all listed names must be from $\mathcal{CN}$.

The semantics of object-oriented action systems is given by a mapping to action systems which is based on the work presented in [9]. The main idea of the mapping is to create one action system per object and join all action systems as specified in the system assembling block.

3.3 UML as Action System

Most UML elements are mapped to corresponding OOAS structures, e.g., classes, member fields, and methods. Transitions of the state machine are roughly mapped to actions. Only the time- and event semantics of UML need to be mapped to more complex OOAS structures (cf. [20]). Let us consider an example where we want to map the transition from state *OpenAndUnlocked* to state *ClosedAndLocked*, see Figure 2. The transition is triggered by the *Close* signal. In UML events are queued in a pool to allow several transitions to be triggered by one signal. If all transitions have been taken, the according event is removed from the pool. We model the event pool as a queue. The following listing shows the mapping of the UML transition to the OOAS semantics:

```
transition_OpenAndLocked_to_ClosedAndLocked =
  requires (state = OpenAndLocked) and
           (events <> [nil]) and
           (hd events)[0] = received_AlarmSystem_Close):
    state := ClosedAndLocked
  end;
/* .. other transitions .. */
dequeue =
  requires events <> [nil] :
    events := tl self.events
  end
```

Here, the **requires** *guard* : *body* construct represents a guarded command *guard* → *body*. The action guard ensures that the source state of the transition is valid, there is some event in the queue, and the head element is the required *Close* event. If the guard is satisfied, the action body updates the state to *ClosedAndLocked*. The actions of the modeled transitions are combined in the **do od** via non-deterministic choice. Entry and exit actions are sequentially composed with transition actions. The following listing shows the composition of actions for our example. The dequeue action, which removes the head element, is executed when no other action is enabled.

```
do
  ( (transition_Armed_to_Alarm;
     call_ShowUnarmed;
     call_ActivateAlarm;
     call_AcousticAlarm_SetOn )
    [] transition_OpenAndLocked_to_ClosedAndLocked
    [] /* .. other transitions .. */
  ) //  dequeue()
od
```

For more details on the UML-OOAS mapping see [20] .

3.4 Trace Semantics of Action Systems

For black-box test case generation purposes, we are interested in the abstract computation sequences, i.e., traces of actions in an action system. In [5], the computation of an action system starting from an initial state γ_0 is defined as a possibly infinite sequence $t =_{df} \gamma_0 \xrightarrow{A_i} \gamma_1 \cdots$ with each guard of A_i enabled in the transition's initial state. Note that this definition also applies to OOAS as we translate them to normal action systems.

The named (labeled) actions define these abstract computation traces. In addition, we extend these names to include markers for *observable* (output) and *controllable* (input) actions. All methods and all unmarked actions are considered *internal*. Hence, any name A^i_n of a named action A_i is built according to the following grammar: $A^i_n ::=$ (*'obs'* | *'ctr'* | *' '*) *' ' Identifier* .

Informally, an abstract computation sequence starting from an initial state γ_0 is a possibly empty or infinite sequence t_{abs} of the form $t_{abs} =_{df} \gamma_0 \xrightarrow{A^i_n} \gamma_1 \cdots$ where $\xrightarrow{A^i_n}$ means the application (call) of the action body A^i_b of action A_i when $g.A^i_b$ holds at the transition's initial state or there is some sequence of basic actions (including method calls) $\gamma_j \xrightarrow{A_i} \cdots$ starting at the current state γ_j and leading to a state where the guard $g.A^i_b$ holds. We denote the alphabet of action names as $L =_{df} \{A^1_n, \dots, A^k_n\}$ where the set of controllable actions L_I and observable actions L_U form a partition: $L =_{df} L_I \cup L_U$ and $L_I \cap L_U =_{df} \emptyset$. Hence, the traces of an action system A are a set of sequences of action names, i.e., $traces(A) \subseteq L^*$.

After translating the OOAS of Figure 2 to a normal action system, a state space exploration of the system yields the labeled transition system (LTS) depicted in Figure 3. This

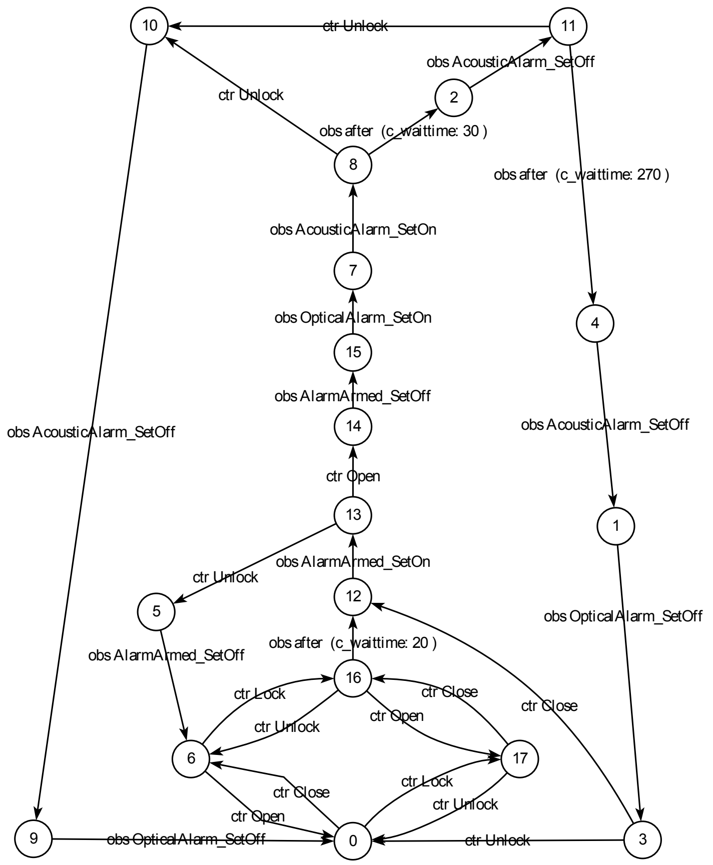

Figure 3: Alarm System - Labeled Transition System.

graph was produced by translating the OOAS to an LTS representation for the CADP tool. Then, in CADP we used the simplifier to exclude internal actions, model checked some properties to validate the model, and finally produced the graph. Having an LTS interpretation of our models, we are able to exploit the established testing theories.

4 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 conformance relation defines if a mutant represents a faulty implementation, i.e., if conformance between the original and the mutant does not hold.

Because we are interested in input-output testing with partial system models we apply the *ioco* relation [25]. It requires that for all traces in the specification over inputs, outputs, and quiescence the following holds: the outputs of the implementation after a trace must be included in the set of outputs of the specification after the same trace. Here, quiescence denotes the absence of observations in a given state, represented as δ self loops. An LTS, augmented with quiescence is called *suspension automaton.*

Consider the example in Figure 4. It shows the partial suspension automata of the specification and a mutant.

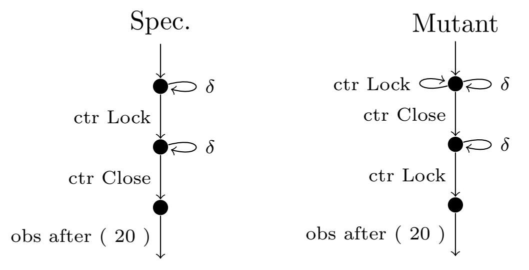

Figure 4: Part of the suspension automata of the specification and a mutant.

In the mutant the controllable action *Lock* is a self loop. One can observe that the mutant is not *ioco* conform to the specification because the subset inclusion of observations *obs* after the trace $\langle Lock, Close\rangle$ does not hold, i.e., $obs(Mutant\ after\ \langle Lock, Close\rangle) = \{\delta\} \not\subseteq obs(Spec\ after\ \langle Lock, Close\rangle) = \{after(20)\}$.

5 Test Generation with Ulysses

Given two UML models (a specification and a mutant), our tool chain computes a set of *ioco* test cases that reveal the mutation, see Figure 5. In a first step, the UML models are translated into object-oriented action systems which our tool *Argos* subsequently translates into action systems. Afterwards, our test case generator *Ulysses* takes the mutated and the original action system and produces mutation-based test cases as output. Figure 6 depicts the computation steps of *Ulysses*.

Our *Ulysses* tool is a model animator and test case generator for hybrid system models. It accepts as input a hybrid variant of action systems, called *Qualitative Action Systems* [2]. Since qualitative action systems are a conservative extension to the (non-oo) action systems presented in this paper, *Ulysses* is also able to create test cases for discrete models. Note that we are on the way of integrating the hybrid systems extensions into our object-oriented action systems framework and that automated test case generation for small qualitative action systems has been presented in [1]. All the results in this paper are based on an improved version of *Ulysses*: besides an interpretation mode *Ulysses* now features a compilation mode that enables the tool to handle larger specifications more efficiently. Currently, this mode is only available to non-hybrid system models. In compilation mode, a given discrete action system model is translated into plain Prolog clauses. The exploration of a compiled action system is at an exponential factor faster than the exploration in interpretation mode.

Ulysses first explores two given action systems yielding their LTS semantics. The actions are executed according to their occurrence in the **do do** block and depending on whether their enabledness guard is satisfied. Then, the LTSs are augmented with δ loops and afterwards converted via subset construction into deterministic automata, called *sus-*

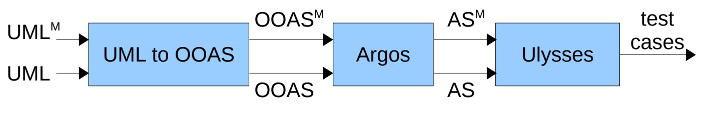

Figure 5: Test Case Generation Tool Chain.

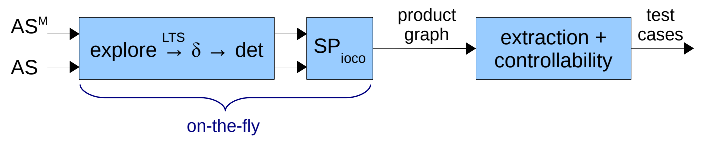

Figure 6: The computation steps of Ulysses.

pension automata. After this, the ioco check generates a product graph as output from which we extract controllable test cases. The extraction is implemented by calculating paths to the differences between the original and a mutated specification, which were identified during product calculation. A new test case for a mutant is only created if no previously generated test case is able to kill it. The calculation of the suspension automata and the synchronous product calculations modulo *ioco* (SP_{ioco}) are performed on the fly which means that the automata are only unfolded as required by the conformance check.

6 Results

To evaluate our generated test cases we used classical mutation testing and compared the results of our mutation-based approach with test results obtained by CADP-TGV [18]. Table 1 gives some numbers about the test suites generated by these two approaches. Note that since the system is deterministic, all test cases are linear.

Our mutation-based test case generator *Ulysses* produced 63 test cases with a maximum depth of 23 actions in 23 minutes from 76 mutated specifications. Our approach is designed not to produce any duplicates, but cannot avoid subsumptions (e.g., prefix inclusion of one test case in another). Hence, the amount of unique test cases is 59.

The reference test suite generated by TGV results from nine different manually designed test purposes. Out of the nine test cases, there are three that test observable timeouts (time-triggered transitions: 20, 30, 300 seconds). Four test cases check the entry and exit actions of the states *Armed* and *Alarm* and one tests the deactivation of the acoustic alarm after the timeout. Finally, one more complex test case, which has a depth of 30 transitions, completes the test suite. It goes once through the state *SilentAndOpen* to *Armed* before it traverses to *Alarm* again, which it leaves after the acoustic alarm deactivation by an unlock event. Note that we relied on a print-out of the UML state machine model (see Figure 2) during the creation of the test purposes.

For the evaluation of the test suites via mutation testing,

		Mutation-based TCG	TGV
UML Mutants	[#]	76	-
Max. Depth		23	30
Gen. TCs	[#]	63	9
Duplicates	[#]	0	0
Unique	[#]	59	9
Gen. Time	[min]	23	-

Table 1: Statistics about generated test cases.

	Mutants	Equivalent	Pairwise Equivalent	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

Table 2: Injected faults into the CAS implementation.

we implemented the CAS in Java according to the state machine in Figure 2 and used μJava [21] to create a set of faulty implementations. In total, we derived 72 mutated implementations, as can be seen in Table 2. After careful inspection, 8 mutants were found to be equivalent and another set of 26 mutants were found to be equivalent to other mutants forming 26 equivalent pairs. Hence, in total 38 different testable faults remain. The events *Close*, *Open*, *Lock*, and *Unlock* are external events while *SetState* and *Constr* are internal events. One can observe that a mutation on internal events has a strong effect on the external behavior since there are no equivalent mutants.

Before applying the test cases to the mutated implementations, all tests were run on a non-mutated implementation and it was checked that no test was failing. Additionally, we verified that the testing results matched between duplicate mutants and that no test failed on any of the equivalent mutants. Table 3 summarizes our testing results by stating the number of mutants that could not be killed. Our mutation-based approach missed to detect one out of the 38 unique mutants resulting in a fault detection rate of 97%. The test cases generated with TGV could not correctly identify 13 of the faulty implementations and thereby achieves a fault detection rate of 66%.

This case study demonstrates the applicability of our mutation-based technique and its good fault detection ability. Compared to TGV's approach, which additionally requires the preparation of test purposes, it detected approximately 30% more of the injected faults. It seems that the test cases derived by manually written test purposes were not diverse enough. Partly, this can be attributed to TGV's deterministic test case selection, which produces tests having the same start sequence.

	Mutation-Based TCG	TGV
SetState	0	0
Close	0	2
Open	1	4
Lock	0	2
Unlock	0	5
Constr.	0	0
Total	1	13
Detection Rate	97%	66%

Table 3: Overview of how many faulty SUTs could not be killed by the generated test cases.

7 Related Research

Automated test case generation from UML state charts has already been considered in [19]. The approach works on an intermediate representation called Testing Flow Graph (TFG), which is basically obtained by flattening the UML state chart. From this TFG, test sequences are selected in order to achieve transition coverage. In contrast to our work, they use fault injection only for test assessment (mutation analysis). A second different lies within the complexity of the used UML state charts: the approach in [19] does not support orthogonality.

The work in [16] deals with test case generation from UML state charts. Test cases are derived due to a test specification language. The approach is founded on the formal semantics of Input-Output LTSs (IOLTSs) and the authors apply a conformance relation similar to ioco [25]. In contrast to our work they do not support time triggers or events with parameters. Additionally we have support for rich data types like lists, sets, and maps and OCL expressions.

Other related research comprises two aspects: the semantic mapping and the model-based mutation testing approach. There exists a UML profile for action systems [26]. This work is exactly the opposite of ours, as it maps action systems to UML.

More closely related is the approach of Petre and Sere on the combined use of state charts and OOAS [22]. The goal is step-wise development via refinement, not testing. Furthermore, only flat state charts are used and the translation is not automated. There has also been work on defining a mapping of UML to B which, according to [15], did not entirely meet the expectations as *schematic translations that attempt to cover a broad class of UML models usually result in B models that are hard to read and quite unnatural*. Since we use the action systems as back-end representation for our test case generator, this is not a problem for us. Again, only simple state charts have been considered.

The rCOS framework uses guarded designs for modeling the behavior of UML components [12]. This semantic framework is similar to action systems. However, rCOS aims for refinement, not for test case generation.

The concept of labeled actions can already be found in [5]. In [11] an event-based view of action systems similar to ours is taken.

Model-based mutation testing was first applied to predicate-calculus specifications [10]. Later, Stocks applied mutation testing to formal Z-specifications [24] without automation. Full automation came with the use of model checkers [4]. The idea was to derive test cases from counterexamples to temporal formulas claiming equivalence between an original and a mutated model. In contrast to this state-based equivalence checking, we check for input-output conformance allowing non-deterministic models. The idea of using an ioco checker for mutation testing is from Weiglhofer who tested against LOTOS specifications [3]. To our knowledge, we are the first who apply ioco checking to UML state machines.

8 Conclusion

In this paper we have presented our results in applying model-based mutation testing to UML state machines. This was the first time that we have presented the whole tool-supported methodology. An industrial car alarm system served as illustrating example. This example is small but non-trivial: it involves hierarchical states, entry-exit actions with methods defined in class diagrams. Experiments showed the fault detecting power of our approach.

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 stateflows, or UML state diagrams" [8]. We agree and see an important contribution of this work in satisfying this requirement. Furthermore, to our knowledge this is the first time that mutated UML state machines have been used for automatic test case generation.

Currently, we are working on larger case studies and refine the tools. Furthermore, we investigate approaches to minimize the needed mutations.

Acknowledgement. Rupert Schlick from AIT (Austrian Institute of Technology) 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

[1] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Model-based mutation testing of hybrid systems. In *Proc. of Formal Methods for Components and Objects (FMCO) 2009*, volume 6286 of *Lecture Notes in Computer Science*. Springer, 2010.

[2] Bernhard K. Aichernig, Harald Brandl, and Willibald Krenn. Qualitative action systems. In *ICFEM '09: Proc. of the 11th Int. Conf. on Formal Engineering Methods*, pages 206–225. Springer, 2009.

[3] Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Protocol conformance testing a SIP registrar: An industrial application of formal methods. In *Proc. of the 5th IEEE Int. Conf. on Software Engineering and Formal Methods*, pages 215–224. IEEE, 2007.

[4] P.E. Ammann, P.E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. of the 2nd IEEE Int. Conf. on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE, 1998.

[5] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3(2):73–87, 1989. Appeared previously in 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing 1983.

[6] Ralph-Johan Back and Kaisa Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.

[7] Ralph-Johan Back and Kaisa Sere. Superposition refinement of parallel algorithms. In *FORTE '91: Proc. of the IFIP TC6/WG6.1 4th Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 475–493. North-Holland Publishing Co., 1992.

[8] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In *Proc. of 15th IEEE Int. Conf. on Automated Software Engineering (ASE2000)*, pages 81–89. IEEE, 2000.

[9] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In *Mathematics of Program Construction, LNCS 1422*, pages 68–95. Springer, 1998.

[10] Timothy A. Budd and Ajet S. Gopal. Program testing by specification mutation. *Computer languages*, 10(1):63–73, 1985.

[11] M.J. Butler and C.C. Morgan. Action systems, unbounded nondeterminism, and infinite traces. *Formal Aspects of Computing*, 7:37–53, 1995.

[12] Xin Chen, Jifeng He, Zhiming Liu, and Naijun Zhan. A model of component-based programming. In *FSEN 2007: International Symposium on Fundamentals of Software Engineering*, volume 4767 of *LNCS*, pages 191–206. Springer-Verlag, 2007.

[13] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[14] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.

[15] Houda Fekih, Leila Jemni Ben Ayed, and Stephan Merz. Transformation of B specifications into UML class diagrams and state machines. In *SAC '06: Proc. of the 2006 ACM Symp. on Applied Computing*, pages 1840–1844. ACM, 2006.

[16] Stefania Gnesi, Diego Latella, and Mieke Massink. Formal test-case generation for UML statecharts. In *ICECCS '04: Proc. of the 9th IEEE Int. Conf. on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age*, pages 75–84. IEEE Computer Society, 2004.

[17] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[18] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.

[19] Supaporn Kansomkeat and Wanchai Rivepiboon. Automated-generating test case using UML statechart diagrams. In *Proc. of SAICSIT 2003*, pages 296–300, 2003.

[20] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. Mapping UML to labeled transition systems for test-case generation – a translation via object-oriented action systems. In *Proc. of Formal Methods for Components and Objects (FMCO) 2009*, volume 6286 of *Lecture Notes in Computer Science*. Springer, 2010.

[21] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: a mutation system for java. In *ICSE '06: Proc. of the 28th Int. Conf. on Software Engineering*, pages 827–830. ACM, 2006.

[22] Luigia Petre and Kaisa Sere. Developing control systems components. In *In Proceedings of IFM'2000: Second International Conference on Integrated Formal Methods, Schloss Dagstuhl, Saarland, Germany*, volume 1945 of *LNCS*, pages 156–175. Springer-Verlag, November 2000.

[23] Emil Sekerinski and Kaisa Sere. A theory of prioritizing composition. Technical Report 5, Turku Centre for Computer Science, 1996.

[24] Philip Alan Stocks. *Applying formal methods to software testing.* PhD thesis, Department of computer science, University of Queensland, 1993.

[25] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[26] Tomi Westerlund and Tiberiu Seceleanu. An UML profile for action systems. Technical Report 581, Turku Centre for Computing Science, December 2003.