Journal of Logical and Algebraic Methods in Programming 85 (2016) 806–823

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Does this fault lead to failure? Combining refinement and input–output conformance checking in fault-oriented test-case generation

Bernhard K. Aichernig [a,*], Elisabeth Jöbstl [b], Martin Tappler [a]

[a] Graz University of Technology, Institute of Software Technology, Austria
[b] AVL List GmbH, Graz, Austria

ARTICLE INFO

Article history:
Received 18 June 2015
Received in revised form 11 February 2016
Accepted 12 February 2016
Available online 17 February 2016

Keywords:
Model-based testing
Test-case generation
Mutation testing
MoMuT
Refinement
Ioco

ABSTRACT

In this paper we describe an advanced test-case generation technique that is implemented in our model-based test-case generator MoMuT::UML. The tool injects faults into a UML model and analyses if the faults propagate to the interface. If a fault does propagate to an observable failure, an explaining sequence of events is generated and converted into a test-case scenario. The faults are detected using a highly optimised refinement checker, their propagation is analysed with an input–output conformance (ioco) checker. We show that this combination is faster than pure input–output conformance checking. It has been used in a recent industrial application of testing automotive measurement devices. The refinement and ioco checker are implemented in Prolog using the SMT solver Z3.

1. Introduction

Recently, Daniel Murta and José Nuno Oliveira [1] pointed out that we see a "trend towards tolerating hardware unreliability, accuracy is exchanged for cost savings. Running on less reliable machines, functionally correct code becomes risky and one needs to know how risk propagates so as to mitigate it. Risk estimation, however, seems to live outside the average programmer's technical competence and core practice." Therefore, they propose that program design by source-to-source transformation be risk-aware and study the fault propagation in functional programs.

Unfortunately, not only programmers, but also testers are often unaware of risk and how faults propagate. Too often the test focus is on covering requirements or code and not on possible faults.

MoMut[1] is a family of model-based test-case generators that implement a fault-oriented coverage strategy [2]. They support different modelling styles but have in common that the generated test cases do not cover a set of model elements, but possible faults injected into the model. The rationale behind this strategy is that the notion of model coverage is highly dependent on the modelling style and level-of-abstraction. For example, Tiran [3] has shown that a car alarm system can be modelled in quite a variety of UML state machines with the number of states ranging from one state (with OCL expressions) to seventeen states (without nested states). Therefore, notions like state coverage or transition coverage become arbitrary

* Corresponding author.
E-mail address: aichernig@ist.tugraz.at (B.K. Aichernig).

[1] http://www.momut.org.

http://dx.doi.org/10.1016/j.jlamp.2016.02.002
2352-2208/

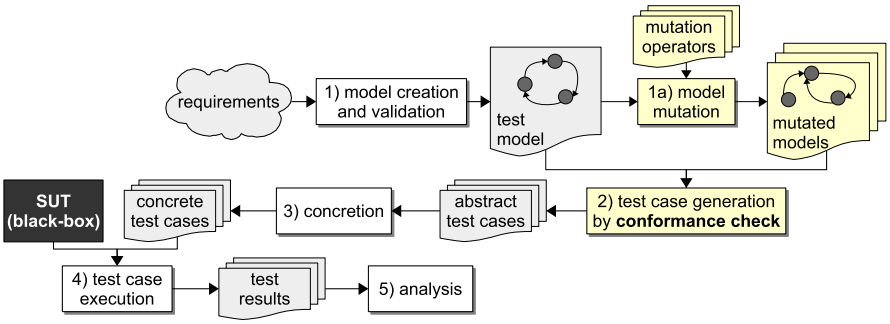


Fig. 1. Overview of model-based mutation testing [6].

and might induce a wrong feeling of trust into the testing process. In the car alarm example, the same test case might cover between 6% and 100% of UML states, depending on the modelling style.

In contrast, the notion of fault coverage is not so much coupled to the modelling style, but to the level of abstraction. The more abstract the model, the less possibilities of introducing faults into it. This means that a given test model does not only reflect the expected behaviour of a system under test (SUT), but also the possible cases of misbehaviour in the form of modelling faults. We will represent these possible misbehaviours as altered models with faults injected into them. In the testing literature these faulty models are called *mutants*, the process of fault injection is known as *mutation*, the rewriting rules that inject the faults are so-called *mutation operators* and the resulting testing process is called *mutation testing*. Its model-based version for generating test cases covering faults is called *model-based mutation testing*. Its main advantage is the direct link between a generated test case and its test purpose: the prevention of a certain type and location of fault in the SUT.

In a fault-oriented testing process, the definition of fault and failure is essential. The IEEE defines a *fault* as an incorrect step, process, or data definition in a computer program [4]. This definition can be applied to behavioural test models, too. If a fault is executed, an *error* might occur which is the difference between a computed and a specified value. A *failure* happens if this error is propagated to the system's external interface and we can observe that it is unable to fulfil its required functions within specified performance requirements.

In this work, we define faults via the state-based notion of operational refinement and failure via the event-based input–output conformance (ioco) of Tretmans [5]. In the context of state-based models, we speak of a faulty model or faulty mutant if it does not operationally refine the original model. This fault propagates to an observable failure if it is not ioco-conform to the original model.

The state-based refinement property is easier to check automatically, since we can decompose the problem using standard refinement laws, i.e., each event (operation) in the model can be checked separately. For our fault-oriented test-case generation approach we are interested how this fault propagates to a wrong output event. Therefore, a more complex ioco-check is added: it involves the interpretation of the model and its mutant as input–output labelled transition systems and forming their synchronous product.

This novel test-case generation technique has been applied in a large industrial case study [6], where we reported on the successful generation of a powerful test suite and its fault-finding capabilities. However, in [6] we only briefly mention the idea of combining refinement and ioco checking without detailed explanations and analysis. In our earlier work on model-based mutation testing we either used standalone refinement-checking [7] or ioco checking [8,9] — never both in combination.

The main contribution of this work is a detailed analysis of the combined refinement and ioco checking approach, comparing it to a stand-alone ioco-checking. It is the first time that we show that the combined approach is faster. The experiments and results form part of the doctoral thesis of E. Jöbstl [10] and have not been published before. To our knowledge this is the first analysis of combining refinement and ioco checking in a test-case generation process.

The rest of the paper is structured as follows: Section 2 explains the model-based testing process, the test-case generator and its associated conformance relations. Next, Section 3 presents the combined conformance checking approach, which is evaluated in two case studies in Sections 4 and 5. Then, these empirical results are discussed in Section 6, before we draw our conclusions in Section 7.

2. Preliminaries

2.1. Model-based mutation testing

Fig. 1 gives an overview of our testing approach, which we refer to as *model-based mutation testing*. It is an integration of mutation testing into the classical model-based testing process.

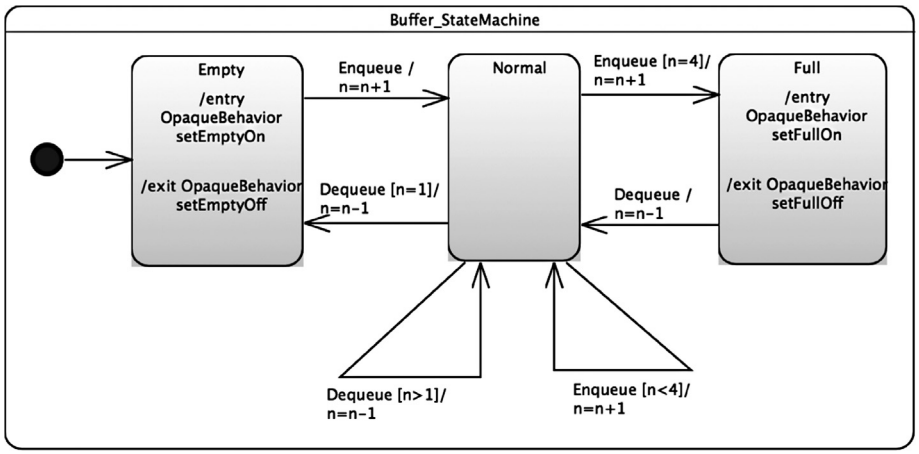


Fig. 2. A test model of an abstract 5-place buffer in form of a UML state machine.

Model-based testing (MBT) is a black-box testing technique requiring no knowledge about the source code of the SUT. Only the interface to the SUT has to be known. A test engineer creates a formal model that describes the expected behaviour of the SUT (Step 1). Test cases are then automatically derived from this test model.

A crucial matter in MBT is the choice of the test criterion. It specifies which test cases shall be generated and hence has a great influence on the quality of the resulting test suite. Exhaustive testing, i.e., using all of the test cases that can possibly be created from the test model, is impractical. Examples for commonly used test criteria are coverage criteria, random traversals, equivalence classes, or specified testing scenarios (test purposes). We follow a fault-centred approach, i.e., use mutations for test case generation (TCG). We syntactically alter the original test model producing a set of mutated models (Step 1a).

We then automatically generate test cases that *kill* the model mutants, i.e., reveal their non-conforming behaviour. This is accomplished by a conformance check between the original and the mutated models (Step 2). Hence, killing a mutant means detecting a non-conformance between the mutant and the original model. The generated test-case leads to this non-conformance, i.e., it kills the mutant. As the test model is an abstraction of the SUT, also the derived test cases are *abstract*. Hence, they have to be concretised, i.e., mapped to the level of detail of the SUT (Step 3).

Finally, the concrete test cases can be executed on the SUT (Step 4) and the test results can be analysed (Step 5). A particular feature of the generated test suites is their fault coverage. The generated tests will detect whether a faulty model has been implemented instead of the correct, original model. Hence, the generated test suite covers all of the modelled faults expressed by the model mutation operators and has a high chance of covering many additional similar faults (cf. coupling effect [11]).

Example 1. Consider the UML state machine of a 5-place buffer in Fig. 2. This test model abstracts away from the concrete buffer contents, but only considers the number of elements in the buffer. Its visible input events are *Enqueue* and *Dequeue*. The internal variable n captures the number of elements in the buffer. The visible output events are modelled as the entry/exit actions *setEmptyOn/Off* and *setFullOn/Off*. The state machine captures the valid input–output behaviour and can thus serve as a source for automated test-case generation. A test case consists of both, input and expected output events. For example, a test case covering all three states would be the sequence $t_1 = \langle$*!setEmptyOn, ?Enqueue, !setEmptyOff, ?Enqueue, ?Enqueue, ?Enqueue, ?Enqueue, !setFullOn*$\rangle$ where we indicate input events with a question mark and output events with an exclamation mark.

Mutation is the process of injecting a single fault into the model. Consider the *Enqueue*-transition from state *Normal* to *Full*. Here, an interesting mutation is replacing the assignment $n = n + 1$ by $n = n$ which is equivalent to deleting this assignment. It models the fault that a programmer got the last update of the counting variable wrong.

This fault is interesting, because it takes several interactions until its effect propagates to a visible failure. The reason is that the wrongly set variable n is internal. For example, the previous test case t_1 passes although it triggers the fault. In model-based mutation testing we generate test cases that detect such injected faults. The following test case is an example of such a test case: $t_2 = \langle$*!setEmptyOn, ?Enqueue, !setEmptyOff, ?Enqueue, ?Enqueue, ?Enqueue, ?Enqueue, !setFullOn, ?Dequeue, !setFullOff, ?Enqueue, !setFullOn*$\rangle$. This test case continues its interaction until a difference in the observations can be noticed. Here, the state *Full* is revisited such that the inconsistency between n and the state *Full* can be detected: an implementation of this faulty model would fail to produce the expected *!setFullOn* output event in the final step of test case t_2. In the following, we describe our tool that automatically generates such fault-detecting test cases from UML state machines.

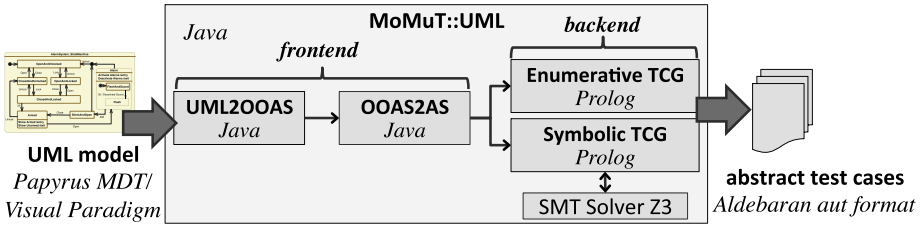


Fig. 3. Architecture of the *MoMuT::UML* tool chain [6].

2.2. MoMuT::UML

Tool support for our model-based mutation testing approach is provided by the *MoMuT::UML* test case generator [2]. It takes a UML model of the SUT, automatically creates mutated models, and subsequently uses these models for the automatic generation of abstract test cases. For model creation, we rely on external UML modelling tools like Visual Paradigm and Papyrus. The concretion and execution of the abstract test cases has also not been integrated in *MoMuT::UML* as these tasks highly depend on the SUT.

Fig. 3 gives an overview of the inputs, outputs, and the architecture. As input, it requires (1) a class diagram and (2) a state machine modelling the SUT. The class diagram defines the input and output events, i.e., the test interface. The state machine defines the possible input–output behaviour as explained in Example 1. There exist plenty of UML modelling tools. Most of them work with a very specific or even proprietary format. Hence, it is not feasible to support all possible UML modelling tools and we concentrated on support for Papyrus MDT (an Eclipse Kepler plugin)[2] and Visual Paradigm for UML 10.2.[3]

Like all model-based testing tools, *MoMuT::UML* delivers *abstract* test cases, which describe sequences of events on the model's level of abstraction. *MoMuT::UML* uses the Aldebaran format.[4] It is a very simplistic and straightforward format for Labelled Transition Systems (LTS), which can be used to represent test sequences. The format is also supported by other tools, e.g., the CADP toolbox.[5]

As shown in Fig. 3, *MoMuT::UML*'s architecture distinguishes between *frontend* and *backend*. The frontend is implemented in Java, the backends are implemented in Sicstus Prolog. Currently, *MoMuT::UML* is available for 64-bit Windows and Linux.

Frontend. The frontend is responsible for converting the UML model into a representation suitable for the backend, i.e., the actual test case generator. First, the UML model is transformed into a labelled and object-oriented action system (OOAS [12]). This executable intermediate representation has a formal semantics and is based on a generalisation of Dijkstra's guarded command language [13] and Back's *action system* [14] formalism. Most UML elements can be directly mapped to the corresponding OOAS structures, e.g., classes, member fields, and methods. Transitions of the state machine are roughly speaking mapped to actions. Only the time- and event semantics of UML need to be expressed by more complex OOAS structures.

Example 2. Fig. 4 shows an action system model of the 5-place buffer of the UML state machine in Fig. 2. For this simple model no object-oriented features are needed. It consists of the counter variable n, the state variable s and an auxiliary variable *pre* storing the previous state (of a transition). The behaviour is modelled with two input (controllable) actions *Enqueue*, *Dequeue* and four output (observable) actions *setEmptyOn*, *setEmptyOff*, *setFullOn*, *setFullOff*. Each input action uses a nested guarded command to express its two cases. In the *do-od* block the actions are composed via non-deterministic choice. This action system iterates over the actions and selects non-deterministically one enabled action. An action is enabled if its guard(s) evaluates to true. If selected, the body of the action is executed.

Note, that this action system model follows the style of the UML model very closely, but it is no automated translation. The generated models are more complex as the translator implements the general semantics of nested UML state machines with parallel regions.

In the next phase of the frontend, the OOAS is flattened to a non-object-oriented, but still labelled action system (AS) the backend(s) can work with. The frontend is also home to the mutation engine that injects faults to construct model mutants. Mutations are inserted at the UML level. The mutation operators are applied to the following state machine elements: triggers, guards, transition effects, entry- and exit actions. The elements are either removed or replaced with another element of the same type from the model. This leads to $O(n)$ mutants for the removals and $O(n^2)$ mutants for the replacements (with n being the number of elements in the model). Additional operators exist for change trigger expressions, for guards

[2] https://www.eclipse.org/papyrus, 18.03.2014.
[3] http://www.visual-paradigm.com/product/vpuml, 18.03.2014.
[4] http://www.inrialpes.fr/vasy/cadp/man/aldebaran.html#sect6, 18.03.2014.
[5] http://cadp.inria.fr, 18.03.2014.

```
types
  Int = int[0..5];
  State = {empty,normal,full};
  Buffer = autocons system
  ||
    var
      n  : Int = 0;
      s  : State = empty;
      pre: State = normal;
    actions
      ctr Enqueue = requires (s <> full):
         requires n < 4: n := n+1; pre := s; s := normal;
         []
         requires n = 4: n := n+1; pre := s; s := full;
      end;

      ctr Dequeue = requires (s <> empty):
         requires n > 1: n := n−1; pre := s; s := normal;
         []
         requires n = 1: n := n−1; pre := s; s := empty;
      end;

      obs setEmptyOn = requires (pre = normal and s = empty): pre := s
      end;

      obs setEmptyOff = requires (pre = empty and s = normal): pre := s
      end;

      obs setFullOn = requires (pre = normal and s = full): pre := s
      end

      obs setFullOff = requires (pre = full and s = normal): pre := s
      end
    do
      Enqueue() [] Dequeue() []
      setEmptyOn() [] setEmptyOff() [] setFullOn() []  setFullOn()
    od
  ||
system Buffer
```

Fig. 4. An action system modelling a simplified 5-place buffer in OOAS notation.

expressed in OCL, for effects, entry-/exit actions, and method bodies. The modifications made here are: exchange operators, modify literals, fix (sub-) expressions to literals. They all lead to $O(n)$ mutants. After all model mutants have been generated, they are converted into action systems similarly to the original UML model. These action systems serve as input for the TCG backend.

Backend. The backends are responsible for test-case generation. In the current release there exist two alternative implementations: an enumerative and a symbolic TCG engine.

The enumerative TCG backend, called *Ulysses*, is a conformance checker for action systems and performs an explicit forward search of the state space. This process yields the labelled transition system (LTS) semantics of the UML models and checks if the mutated model conforms to the original model. Conformance is defined via a conformance relation that compares two models. The conformance relation of *Ulysses* is ioco.

The symbolic TCG backend combines a symbolic refinement check with a follow-up ioco check. This input–output conformance check is an optimised re-implementation of *Ulysses*. It can also be started without a prior refinement check. We call this mode of operation *stand-alone ioco check*. All the experiments in this paper have been performed with the symbolic backend.

Before we are going to explain the combined conformance checking approach, the two relevant conformance relations will be introduced in the following.

2.3. Refinement

In our theoretical work on mutation testing, we gave test cases a denotational semantics and related them via refinement to models and implementations [15,16]. This enabled us to formally prove our test-case generation algorithms correct. These results can be carried over to action system models: following the style of He and Hoare [17], we gave action systems a relational predicative semantics [7]. That means that the transition relation is expressed via a predicate denoting the possible observations before and after execution of an action. Here, the observations are the values of the state variables.

For illustration purposes, Fig. 5 shows those parts of the predicative semantics definitions that are relevant to Example 2. First, the non-deterministic choice of two actions is defined as the disjunction of their predicative semantics. Next, an action with a *label*, a guard g and an action body B is defined via conjunction. In an auxiliary variable *trace* the *label* of

$$
\begin{array}{lll}
Action_1 \ [\ ]\ Action_2 & =_{df} & Action_1 \ \vee \ Action_2 \\
label = \text{requires } g : B & =_{df} & g \ \wedge \ B \ \wedge \ trace' = trace \frown [label] \\
\text{requires } g : B & =_{df} & g \ \wedge \ B \\
x := e & =_{df} & x' = e \ \wedge \ y' = y \ \wedge \ldots \wedge \ z' = z \\
B_1(\overline{v}, \overline{v}'); B_2(\overline{v}, \overline{v}') & =_{df} & \exists \ \overline{v}_0 : B_1(\overline{v}, \overline{v}_0) \ \wedge \ B_2(\overline{v}_0, \overline{v}')
\end{array}
$$

Fig. 5. Excerpts of the predicative semantics of action systems.

the selected action is added to the history of past action labels. The test-case generation algorithms use this *trace* in order to generate a test-case. Similarly, inner guarded commands are defined via conjunction. Then, the assignment statement in a body is defined as usual: the assignment variable is set to the value of expression e and the other variables keep their values. Finally, the sequential composition of two statements with predicate semantics is shown: both are predicates over the free prestate variables $\overline{v}$ and poststate variables $\overline{v}'$. The result is a new predicate describing the sequential transition from pre-state to postate via an intermediate state $\overline{v}'_0$.

In this predicative semantics refinement is simply defined via implication:

Definition 1 *(Refinement).* Given two action system models M and O, both with predicative semantics $M(\overline{v}, \overline{v}')$ and $O(\overline{v}, \overline{v}')$ with $\overline{v} = \langle x, y, \ldots \rangle$ being the set of variables denoting observations before execution and $\overline{v}' = \langle x', y', \ldots \rangle$ denoting the observations afterwards. Then

$$M \text{ refines } O \ =_{df} \ \forall \overline{v}, \overline{v}' : \ M(\overline{v}, \overline{v}') \ \Rightarrow \ O(\overline{v}, \overline{v}')$$

We developed a mutation testing theory based on this notion of refinement [16]. The key idea is to find test cases whenever a mutated model M does not refine an original model O, i.e., if $\neg(M \text{ refines } O)$. Hence, we are interested in counterexamples to refinement. From Definition 1, it follows that such counterexamples exist if and only if implication does not hold:

$$\exists \overline{v}, \overline{v}' \ : \ M(\overline{v}, \overline{v}') \wedge \neg O(\overline{v}, \overline{v}')$$

This formula expresses that there are observations in the mutant M that are not allowed by the original model O. We call a pre-state, i.e., a valuation of all variables, *unsafe*, if such an observation can be made.

Definition 2 *(Unsafe state).* A pre-state $\overline{u}$ is called an unsafe state if it may show wrong (not conforming) behaviour in a mutated model M with respect to an original model O. Formally, we have:

$$\overline{u} \in \{\overline{v} \mid \exists \, \overline{v}' : M(\overline{v}, \overline{v}') \wedge \neg O(\overline{v}, \overline{v}')\}$$

We see that an unsafe state can lead to an incorrect next state. In model-based mutation testing, we are interested in generating test cases that cover such unsafe states. Hence, our refinement checker searches for unsafe states. Our implementation uses a breadth-first search through the state-space until an unsafe state is found. The next states are calculated via solving the transition relation of the action systems with the SMT solver Z3. It can be seen as a constrained reachability problem. Hence, its complexity is NP-complete [18]. For details of this search including several optimisations for action systems, we refer to [7].

2.4. Input–output conformance

Refinement is a state-based conformance relation. Its observations are the variable valuations before and after execution. However, the points of observations from a tester's perspective are input and output events. A test case for such systems is a sequence of input and output events in the deterministic case. For non-deterministic systems, test cases have to branch over all possible outputs. Such tree-like test cases are known as adaptive test cases. The operational semantics of such systems is usually given in terms of Labelled Transition Systems (LTS) with (disjoint) label sets representing input, output, and internal events.

We can give action systems such a labelled transition system semantics: the labelled actions with instantiated parameters represent events, the states are the variable valuations, the transitions are defined via the predicative semantics of actions.

The corresponding event-based conformance relation is Tretmans' input–output conformance (ioco) relation [5]. The conformance relation *ioco* is defined as follows.

Definition 3. Given a weakly input-enabled action system M and an action system O, both with LTS semantics with input, output and internal events, their ioco conformance is defined via a subset relation over outputs:

$$M \text{ ioco } O =_{df} \forall \sigma \in \text{Straces}(O) : \text{out}(M \text{ after } \sigma) \subseteq \text{out}(O \text{ after } \sigma)$$

Here after denotes the set of reachable states after a trace σ, and out denotes the set of all output events in a set of states. In addition to standard output events, the absence of any output, i.e., quiescence, is added as special output event.

```
types
  SmallInt = int [0..2];
  CoffeeMachine = autocons system
  ||
    var
      state: SmallInt = 0
    actions
      ctr button = requires state = 0 : state := 1 end;

      obs coffee = requires state = 1 : state := 0 end;
    do
      button() [] coffee()
    od
|| system CoffeeMachine
```

Fig. 6. An action system modelling a simple coffee machine.

The set *Straces* is the set of all possible traces plus additional traces including these quiescence observations. Weak input-enabledness means that M always accepts all inputs (possibly after some internal transitions).

In our application of ioco conformance checking, M is the mutated action system and O the original. In the original definition of ioco, M is a model of the implementation and O is a partial specification model. Hence, the original motivation of ioco was to define conformance between a model and a system under test. Here, we use ioco to define the conformance of a mutant to its original action system model with respect to its input–output behaviour.

This input–output conformance relation *ioco* supports non-deterministic models (see the subset relation) as well as partial models (only traces of O are tested). For input-complete action systems, meaning that all inputs are enabled in all states, *ioco* is equivalent to trace-inclusion (language inclusion) [5].

It is known that the problem of checking ioco is PSPACE-complete [19]. An explicit ioco checker essentially builds a synchronous product of the two action systems and searches for an output label in the mutated model that is not allowed by the specification model. Our implementation of the ioco check is based on the same algorithm as implemented in the ioco-conformance checker *Ulysses* [8,9]: the synchronous product of the two underlying LTS is computed on the fly. That is, the LTS of the mutated and the original action system are explored in parallel and immediately checked for ioco conformance. If non-conformance is detected, the exploration of the LTS is stopped.

The difference to *Ulysses* is as follows. Our exploration of the action systems is based on constraints representing the transition relation. In contrast, our explicit backend *Ulysses* enumerates all possible parameter valuations and then tests whether these values fulfil the guard of a given action. For large domains, this is inefficient and our constraint-based approach usually performs better – particularly with respect to memory consumption.

In the following we present how these two conformance checking algorithms can be combined into an efficient conformance checker.

3. Combining refinement and IOCO checking

3.1. Faults vs. failures

Refinement does not distinguish between internal states and external observations. By definition, any variable assignment that is not allowed by the original specification will lead to non-refinement, i.e., a fault. Hence, the refinement checker stops and produces a test case as soon as an *unsafe state* is detected. This may result in too short test cases as the following small example illustrates.

Example 3. Consider the action system of a coffee machine in Fig. 6. It is modelled with one state variable *state* possibly ranging between 0 and 2, one controllable action for the input event of pressing the *button*, and one observable action representing the event of *coffee* output. The corresponding LTS is depicted on the left-hand side of Fig. 7. The labels of the LTS states correspond to the valuations of the state variable *state*. Consider a mutated version of this action system which redefines the assignment of action *coffee* to $state := 2$, i.e., the post-state of the action is wrongly set to 2 instead of 1. This yields the second LTS named *mutant* in Fig. 7. We see that this mutation causes a termination of the coffee machine: after one coffee no further coffee can be produced.

Our refinement checker identifies State 1 as unsafe state leading to the fault and produces an event-based test case. This *refinement test case* is shown in Fig. 7. It can be easily seen that this test case is too short in the sense that it reaches the fault, but does not lead to a visible failure when executed.

Our ioco checker does exactly this. It explores both LTSs until a wrong observable event is detected. For our example, the ioco checker generates the *ioco test case* in Fig. 7 which tries to produce two coffees. If the SUT behaves as the mutant, the fault will propagate to the failure that no second coffee is produced. Technically, this corresponds to the quiescent State 2 in the mutant, where no output is produced. In practice, the test execution will wait for a defined time-out period and then report the failure.

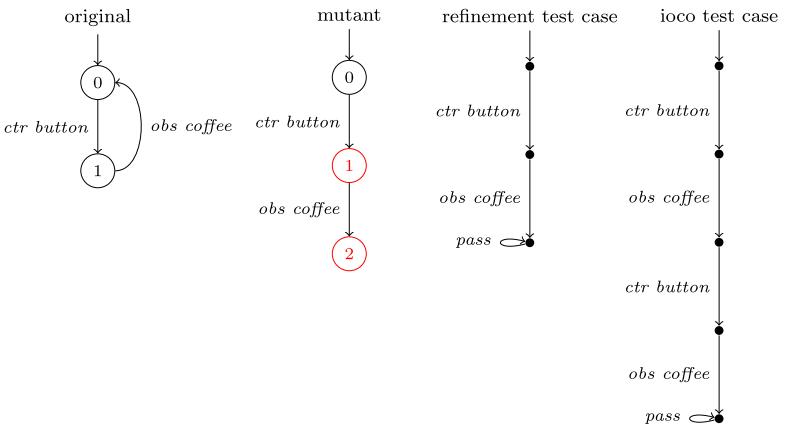


Fig. 7. Our refinement check results in a too short test case for distinguishing the mutant from the original. The ioco checker explores the two LTSs until the fault propagates to a wrong observation.

Example 4. Let us reconsider our motivating buffer Example 1 in the context of action systems. Similar to the discussed UML mutation, we can mutate the action system in Fig. 4 by deleting the assignment $n := n+1$ in Line 14. The refinement checker will explore the system until it arrives in an unsafe state $n = 4$, $s = normal$, $pre = normal$ where refinement is violated by the next mutated *?Enqueue* action. The value of the auxiliary trace variable tr' that is part of the predicative semantics (see Fig. 5) represents the test case triggering this fault: $tr' = \langle$*!setEmptyOn, ?Enqueue, !setEmptyOff, ?Enqueue, ?Enqueue, ?Enqueue, ?Enqueue*$\rangle$. During test-case construction the expected output *!setFullOn* would be appended to the test case. Like in the previous Example 3 this test case is too short. It does not propagate the fault to a visible failure. In contrast, the ioco checker explores the two action systems until the mutant produces an output event that is not allowed by the original model: our tool produces the test case $t_2 = \langle$*!setEmptyOn, ?Enqueue, !setEmptyOff, ?Enqueue, ?Enqueue, ?Enqueue, ?Enqueue, !setFullOn, ?Dequeue, !setFullOff, ?Enqueue, !setFullOn*$\rangle$ of Example 1.

Another problem of refinement is its strictness. Any difference in state may produce test cases although the external observational behaviour, i.e., the event view, is equivalent. Our combined approach solves this problem, because it additionally checks if a difference in state leads to a violation in the input–output conformance.

3.2. Combining refinement and IOCO

We experienced that a full ioco check of two action systems can be rather costly. This can also be seen from our experimental results, which we present in the next sections.

To counteract, we combine our strict, but efficient refinement check with an ioco check. More precisely, we first perform a refinement check. Only if non-refinement is identified, we append an ioco check starting from the unsafe state identified by our refinement check. In this way, the ioco check is more targeted to those parts of the system that are actually affected by the mutation. This allows for higher exploration depths and hence longer test cases.

Example 5. Reconsider the LTS of the action system and its mutation in Fig. 7. The refinement check would identify the state mutation and report the unsafe State 1. Then, the ioco check would take State 1 as initial state and continue building the synchronous product until the difference can be observed. Finally, the test case shown on the right-hand side of Fig. 7 is generated. It covers the fault and ensures that the fault propagates to a (visible) failure.

Algorithm 3.1 *combinedRefIocoTcg*(*as*, *init*, *mutants*, *maxRef*, *maxIoco*).

```
1:  states := findAllStates(as, init, maxRef)
2:  for all asm ∈ mutants do
3:      (u, tr2unsafeRef) := checkRefinement(states, as, asm)
4:      if u ≠ nil then
5:          tr2UnsafeIoco := checkIoco(as, asm, u, maxIoco)
6:          if (tr2UnsafeIoco ≠ nil) then
7:              tr2Unsafe := tr2UnsafeRef ⌢ tr2UnsafeIoco
8:              constructSaveTc(as, init, tr2Unsafe)
9:          end if
10:     end if
11: end for
```

Our combined refinement and ioco test case generation is sketched in Algorithm 3.1. As input, it requires the original action system *as*, its initial state *init*, and a corresponding set of mutated action systems *mutants*. Furthermore, the maximum depths for the refinement check (*maxRef*) and the ioco check (*maxIoco*) need to be specified by the user. In our most optimised refinement checker, we pre-compute the state space up to the given depth for the refinement check in Line 1. This has advantages when many mutants have to be compared to the same original model as the state space can be kept in memory. Then, we iterate over the mutants (Line 2). Each mutant is checked for refinement in Line 3. If non-refinement is detected, the function *checkRefinement* returns the unsafe state *u* together with a trace leading to this state (*tr2UnsafeRef*). If the mutant *asm* refines the original *as* up to the given depth, *nil* is returned for the unsafe state and the next mutant is analysed. If an unsafe state *u* has been found (Line 4), we perform an ioco check starting at the unsafe state *u* with a maximum depth specified by the user. The function *checkIoco* returns a trace that leads from the unsafe state *u*, which was determined by the refinement check, to the unsafe state of the ioco check. If the mutant is ioco-conform to the original (up to the given depth *maxIoco*), the returned trace is *nil*. In this case, no test case is generated and the next mutant is processed. If the trace to the unsafe state of the ioco check is unequal to *nil* (Line 6), it is appended to the trace of the refinement check forming the overall trace to the unsafe state (Line 7). This trace starts at the initial state of the action system and leads to the unsafe state, which triggers an ioco difference. It serves as the basis for a test case (Line 8). The function *constructSaveTc* performs the test case construction. It takes the complete trace to the unsafe state, the original action system, and its initial state. This function adds the test verdicts to the trace. It is especially important for non-deterministic models. The function traverses the trace and adds in each step alternative outputs that are allowed by the model but not included in the trace. These additional outputs represent correct system responses that deviate from the trace to be followed. Therefore, they are marked as being inconclusive. The resulting test case is saved in a file. Note that the resulting test cases may have a length of up to $maxRef + maxIoco$, i.e., the sum of the maximum depths of the two conformance checks.

3.3. Under-approximation

This combination of our refinement check with a subsequent ioco check results in a notion of conformance that is slightly weaker than ioco. Our refinement check classifies certain mutants as refining, although they are not ioco-conform. In this way, also our combination of refinement/ioco classifies such mutants as conforming and does not generate a test case, where a pure ioco check would result in a test case. Therefore, our combined approach results in an under-approximation of the test suite that would be generated by a pure ioco check given the same set of mutants. In the following, we discuss the three cases where non-conformance with respect to ioco is not detected by our combined approach. For illustrative examples we refer to [10].

Quiescence. In our refinement relation, we neglect quiescence (absence of output) as it is not encoded in our predicative semantics for action systems. For the ioco check, it is added during the exploration of the underlying LTS as an additional observation. As a consequence, an implementation may conform to a specification with respect to our refinement relation, but not with respect to ioco. In this way, our combined refinement/ioco check classifies certain mutants as conforming, although ioco does not hold.

Input-enabledness. Furthermore, the differences between our combined refinement/ioco check and a pure ioco check are partly caused by the ioco-assumption that implementations are input-enabled. For ioco, implementation models are implicitly made input-enabled by angelic completion, i.e., by adding self-loop transitions labelled by inputs that are not enabled in a state. In this way, unknown inputs are always accepted, but ignored by implementations. This angelic completion happens only in the ioco check and not in the refinement check, because refinement does not differentiate between input and output. These added inputs might make a mutant non-ioco, but refinement may hold — another case, where we miss test cases in the combined approach.

Stop at first unsafe state. Finally, our refinement/ioco check might wrongly classify mutants as conforming due to the stopping criterion of our refinement check. Our refinement check searches for only one unsafe state with respect to refinement and then performs an ioco check starting at this state. If this ioco check does not find violations of ioco, then the mutant is classified to be conforming. However, it is possible that there exist further unsafe states with respect to refinement that actually propagate to an ioco difference. Hence, to avoid this problem we would have to backtrack, search for another unsafe state with respect to refinement and check if this one propagates. This iterative process stops either if a violation of ioco can be identified, or if no further unsafe states can be found by our refinement checker.

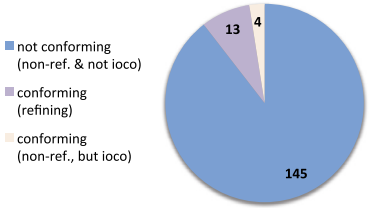
Despite these limitations, our approach seems to be a good trade-off between computation time and fault coverage. It is significantly faster than a stand-alone ioco check as will be seen from our experimental results that we describe in the following.

4. Car Alarm System

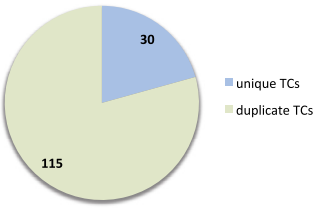
To evaluate our combined refinement/ioco checker, we first applied it to a relatively simple Car Alarm System (CAS) model: the model CAS_UML is an action system that has been translated by our tool MoMuT::UML from a UML state machine.

Table 1
Metrics for the CAS modelled in UML.

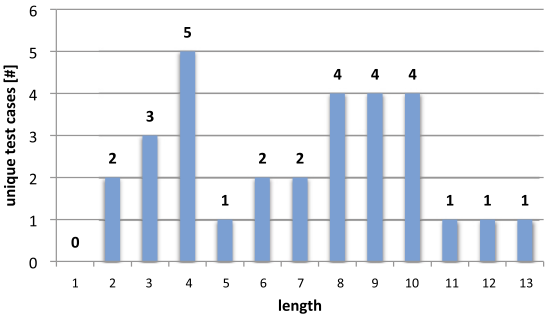
	CAS_UML
actions [#]	51
state variables [#]	35
possible states [#]	$1.7 \cdot 10^{18}$
reachable states [#]	229
required exploration depth	17



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Fig. 8. Test case generation with our combined refinement and ioco check for CAS_UML.

Table 1 states model metrics on the action system level. CAS_UML defines 51 actions, 41 of them are internal actions. The approximate complexity of the models can be estimated by considering the types of the state variables, which are either Boolean or bounded integers. CAS_UML comprises 35 state variables that theoretically build a state space of $1.7 \cdot 10^{18}$ states. However, not all of these states are reachable from the initial system state. In CAS_UML, 229 states are actually reachable. The depth in terms of the number of consecutive visible actions required to reach all of these states is 17.

We conducted a stand-alone ioco check for the model and compared it to our combined refinement/ioco check. All experiments were performed on a computer with an Intel i7 quad-core processor (3.4 GHz) equipped with 8 GB RAM running a 64-bit Linux distribution (Ubuntu 12.04).

Mutations. As explained in Section 2.2, the tool MoMuT::UML provides a rich set of mutation operators for UML state machines. For our CAS_UML model, the following mutation operators were applied: setting guards to false resulted in 22 mutations, removing AGSL[6] statements caused 13 mutations, and removing entry and exit actions resulted in 1 mutant each. By removing effects from transitions, 7 mutations were created. Further 42 mutations were caused by replacing effects with other effects in the model. Furthermore, 14 mutations were generated by removing signal triggers and 3 mutations were caused by removing time triggers. The replacement of signal events caused 42 mutations, and mutating time events resulted in 17 mutations. In total, 162 mutated models have been created.

4.1. Combined refinement and IOCO check

For the CAS_UML model, we set the exploration depth limit of the refinement check to 20 steps. The exploration limit of the ioco check starting in an unsafe state was also set to 20 steps. As the model can be fully explored within 17 steps from the initial state (see Table 1), these settings ensure that the whole model is covered.

Fig. 8 gives an overview of the results from our combined refinement/ioco check. Fig. 8a illustrates that the majority of all model mutants was non-conforming (90%). For the other 10%, conformance was either caused by refinement (13

[6] Activity and Guard Specification Language.

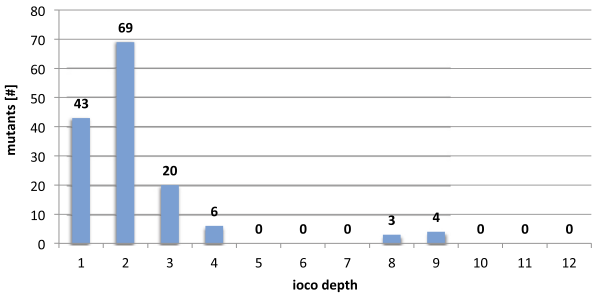


Fig. 9. Overview of the ioco depths for CAS_UML.

mutants) or by the fact that non-refinement did not induce an ioco difference (4 mutants). Non-refinement propagated to an ioco violation in most cases. Thereby, 145 test cases were generated, whereof 115 are duplicates of others (79%). The final test suite consists of 30 unique test cases (Fig. 8b). Fig. 8c shows the lengths of these 30 test cases. The longest test case consists of 13 consecutive actions.

For the non-conforming mutants (non-refining and not ioco-conform), Fig. 9 illustrates the depths required by the ioco check from the unsafe state, i.e., the depths required for the propagation of non-refinement to an observable failure with respect to ioco. The required depths are below 5 steps for the majority of the mutations. Only 7 mutations required a depth of 8 and 9 respectively.

Table 2 gives detailed information about the computation times required for our combined refinement and ioco check. It splits the total computation time into those parts spent on conforming and not conforming mutants, which are restated in the first row of the table. For conforming mutants, it furthermore distinguishes between those that refine the original and those that are non-refining, but did not propagate. The table gives computation times for all mutants considered in a column (Σ), for the average per mutant (ϕ), and the maximum time per mutant (max). We state run-times for the following tasks:

1. The time required for finding all reachable states. Note that this is performed once for all mutants and cannot be split between conforming and non-conforming mutants.
2. The time required for the refinement checks.
3. The time for the ioco checks.
4. The time used for test case construction, i.e., the time to make a test case out of the traces to the unsafe states. It includes the time for writing the test cases into files on the hard drive.
5. Finally, we state the total computation time. It comprises the time for both conformance checks and the time used for test case construction. However, it does not include the time for finding the reachable states as this is done once and is reused for all mutants. Hence, it cannot be split between conforming and non-conforming mutants. Furthermore, we use two categories for the total computation time: one with activated logs and one without log files. Our tool writes comprehensive log files, which can be used for debugging and our evaluations. However, this is a considerable overhead for such small examples and is not required for test case generation in practice, where it can be deactivated.

Note that the reported computation times in all reported experiments refer to the backend activities only. The time for mutating the UML models and translating them to Action Systems is not included.

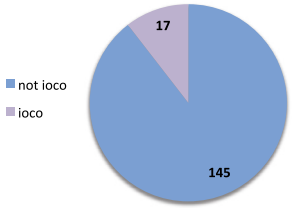
The time used by the refinement checks is shorter than the time required for the ioco checks (1 vs. 2.2 minutes). The time used by test case construction is smaller than the total time for the combined conformance checks (1.3 minutes vs. 3.2 minutes). Our log files cause a substantial amount of the run-time (59%). Overall, the test case generation takes 4.6 minutes if no log files are produced.

4.2. Stand-alone IOCO check

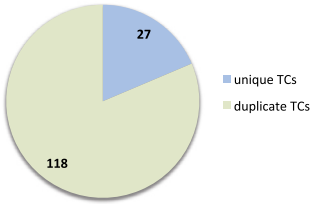
For comparison, we also performed a pure ioco check from the initial state. We specified a maximum search depth of 20, which is sufficient to fully cover this model. In this way, we have a comparable setting to our combined conformance check, where we used $20 + 20$ steps for the search depths. Fig. 10 summarises the results: It shows that 17 mutants were ioco-conform to the original model (10%). Each of the remaining 145 mutants produced a test case. Thereof, only 27 test cases (19%) remained after removing the duplicates (Fig. 10b). The unique test cases show lengths up to 13 consecutive steps (Fig. 10c). Compared to our combined refinement/ioco check, the generated test cases have the same maximum lengths. Also, the number of test cases is almost equal: 27 test cases were generated by the ioco check and 30 test cases by our combined conformance check. Furthermore, both conformance checks resulted in the same set of conforming mutants, i.e., for this model and the given model mutants, we did not under-approximate, but correctly identified all non-conforming mutants. We took a deeper look into the model and some mutations and found out that this model assigns many intermediate results

Table 2
Computation times of test case generation via the combined refinement/ioco check for the CAS_UML model. All values are given in seconds unless otherwise noted.

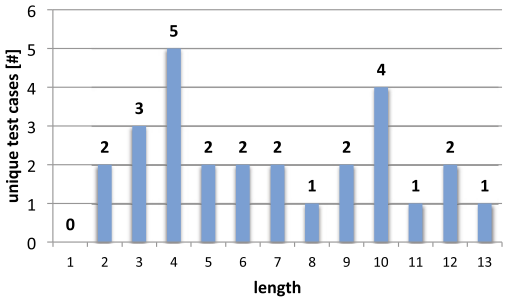
		conforming (refining)	conforming (non-ref., but ioco)	not conforming (non-ref. & not ioco)	total
mutants [#]		13	4	145	162
find states	Σ	–	–	–	3
ref. check	Σ	4.03	1.63	56.41	1 min
	ϕ	0.31	0.41	0.39	0.38
	max	0.41	0.44	0.53	0.53
ioco check	Σ	–	17.71	1.9 min	2.2 min
	ϕ	–	4.43	0.79	0.81
	max	–	4.48	2.01	4.48
tc constr.	Σ	–	–	1.3 min	1.3 min
	ϕ	–	–	0.55	0.49
	max	–	–	1.48	1.48
total	Σ	35.97	28.54	10.1 min	11.2 min
	ϕ	2.77	7.14	4.20	4.16
	max	3.29	7.23	7.53	7.53
total	Σ	4.25	19.4	4.2 min	4.6 min
without log	ϕ	0.33	4.85	1.74	1.7
	max	0.43	4.89	2.77	4.89



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Fig. 10. Test case generation based on a stand-alone ioco check for CAS_UML.

in state variables. Thereby, most mutations involve a difference in the states of the mutated models, which is detected by our refinement check.

Table 3 gives information on the computation times required by the ioco check. The time required for ioco checking amounts to 3.7 minutes, while the time for test case generation is approximately 1.4 minutes. The overall computation time including the time consumed by producing log files is almost 8 minutes. For this model, the time required by writing statistics into log files is not the main part of the overall time. Without log files, the run-time reduces to 5 minutes. What is also worth mentioning, is that the average time required by the ioco check for a conforming mutant is higher than the average time for a non-conforming mutant (3.5 vs. 1.1 seconds). The fact that equivalent mutants raise the overall computation time for ioco checking dramatically has already been reported in an earlier study on explicit ioco checking [9].

Table 3
Computation times of test case generation via ioco checking up to depth 20 for the CAS_UML model. All values are given in seconds unless otherwise noted.

		not ioco	ioco	total
mutants [#]		145	17	162
time – ioco check	Σ	2.7 min	59.97	3.7 min
	ϕ	1.11	3.53	1.36
	max	2.52	4.84	4.84
time – tc constr.	Σ	1.4 min	-	1.4 min
	ϕ	0.57	-	0.51
	max	1.47	-	1.47
total computation time	Σ	6.5 min	1.3 min	7.8 min
	ϕ	2.69	4.61	2.89
	max	4.53	6.21	6.21
total computation time without log	Σ	4.1 min	59.97	5.1 min
	ϕ	1.69	3.53	1.88
	max	3.54	4.84	4.84

Table 4
Metrics for the Particle Counter modelled in UML (PC_UML).

	PC_UML
actions [#]	109
state variables [#]	74
possible states [#]	$1.2 \cdot 10^{31}$
reachable states [#]	$> 850\,700$
required exploration depth	> 25

5. Automotive measurement device

As a second case-study we evaluated the test-case generation algorithms on the model of an automotive measurement device that measures particles in exhaust gases. Again, the model has been translated by our tool MoMuT::UML.

Table 4 states model metrics on the action system level. PC_UML defines 109 actions, 83 thereof are internal. PC_UML requires 74 state variables resulting in a state space of $1.2 \cdot 10^{31}$ states from which more than 850 000 states are reachable. We can only give a lower bound for the number of actually reachable states as it was not possible to fully explore this model. The exploration up to depth 25 took approximately 4 days. A deeper exploration would require even longer run-times. Therefore, we stopped at depth 25.

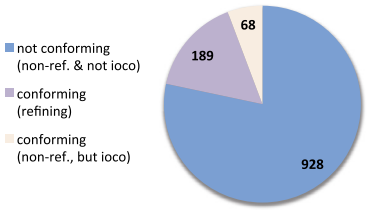
Mutations. The UML state machine for the particle counter has been mutated with MoMuT::UML yielding 1185 mutated models. The following mutation operators were applied. Guards were set to true and false yielding 39 and 23 mutations respectively. Furthermore, the following elements were removed from the model: AGSL statements (23 mutations), change triggers (1 mutation), effects (14 mutations), entry actions (11 mutations), exit actions (5 mutations), signal triggers (27 mutations), and time triggers (6 mutations). The following elements were replaced by other elements of the same type: effects (182 mutations), entry actions (110 mutations), exit actions (20 mutations), and signal events (506 mutations). Furthermore, time events were mutated (36 mutations), e.g., by incrementation by 1. Finally, the exchange of OCL operators caused 29 mutations, and the mutation of OCL subexpressions yielded 153 mutations.

5.1. Combined refinement and IOCO check

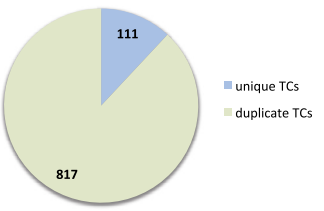
Due to the complexity of the PC_UML model generated by MoMuT::UML's frontend, we restricted the depth for the refinement check to 15 and the maximum depth for the subsequent ioco check to 5. Hence, the model is explored up to depth 20, but only reveals faults that lie within depth 15. Fig. 11 shows the results of our combined refinement and ioco check. As can be seen from Fig. 11a, 189 model mutants refine the original and 68 mutated models are conform as the found non-refinement does not propagate to an ioco violation. The remaining 78% of the mutants are non-conforming and result in 928 test cases. The vast majority of these test cases can be removed as they are duplicates of others. Hence, the test suite consists of 111 unique test cases. The lengths of these test cases are depicted in Fig. 11c. The longest test case comprises 16 consecutive actions.

Fig. 12 illustrates the number of steps required by the ioco checks to identify an ioco violation after the non-refinement. In most cases, a depth of 1 or 2 is sufficient. A depth of 3 is still required by 44 model mutants. Some were also found at depths 4 or 5.

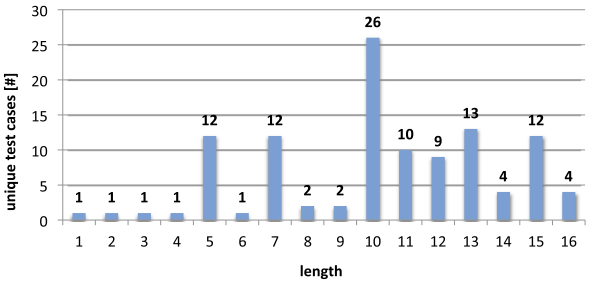
Table 5 summarises the computation times for our combined conformance check. In this setting, the refinement check consumes the majority of the run-time. While 13.3 hours are spent on refinement checking, only 2.4 hours are used for the



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Fig. 11. Test case generation with our combined refinement and ioco check for PC_UML.

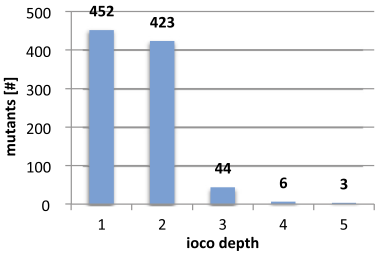
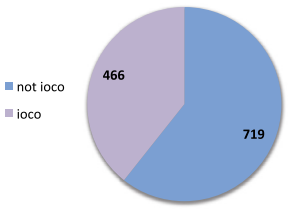


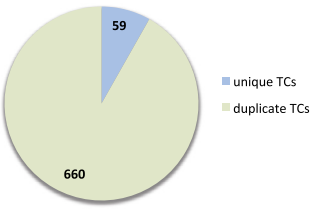
Fig. 12. Overview of the ioco depths in our combined refinement/ioco check for PC_UML.

Table 5
Computation times of test case generation via the combined refinement/ioco check for the PC_UML model. All values are given in minutes unless otherwise noted.

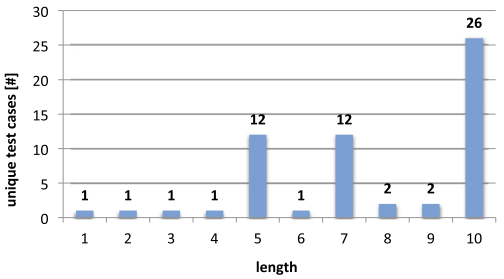
		conforming (refining)	conforming (non-ref., but ioco)	not conforming (non-ref. & not ioco)	total
mutants [#]		189	68	928	1185
find states	Σ	–	–	–	13.5
ref. check	Σ	6.1 h	7.7	7.1 h	13.3 h
	ϕ	1.9	6.8 sec	27 sec	40 sec
	max	4.3	1.8	3.9	4.3
ioco check	Σ	–	0.7 h	1.7 h	2.4 h
	ϕ	–	38 sec	7 sec	7.4 sec
	max	–	2	27 sec	2
tc constr.	Σ	–	–	22.9	22.9
	ϕ	–	–	1.5 sec	1.2 sec
	max	–	–	3.7 sec	3.7 sec
total	Σ	6.2 h	0.9 h	9.7 h	16.8 h
	ϕ	2	0.8	0.6	0.9
	max	4.4	2.2	4.1	4.4
total without log	Σ	6.1 h	0.9 h	9.2 h	16.2 h
	ϕ	1.9	0.8	0.6	0.8
	max	4.3	2.2	4.1	4.3



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Fig. 13. Test case generation based on a stand-alone ioco check for PC_UML.

ioco check. However, for refinement we allowed for a search depth of 15, while the ioco check was restricted to a depth of 5. The overall computation time amounts to 16.2 hours.

5.2. Stand-alone IOCO check

The stand-alone ioco check was performed with a maximum depth of 10. We experienced performance problems with ioco depths higher than 5 for the combined conformance check. However, we can allow for this depth in the pure ioco check, which always starts from the initial state. The reason is that the particle counter model is very narrow in the beginning. The model starts with a sequence of three actions for reporting its state. Only then, the state space slowly widens. For our combined approach, the ioco check starts in deeper states and has to cope with a relatively larger branching factor, i.e., each state has many successor states.

Fig. 13 summarises the results of the ioco check. For this model, the number of ioco-conform mutants is rather high (39%). This can partly be explained by the fact that we do not fully cover the model's state space. The remaining non-conforming mutants result in 719 test cases (cf. Fig. 13a). Thereof, 92% are duplicates and only 59 unique test cases remain (Fig. 13b). The majority of these test cases are 10 steps long (Fig. 13c). This is also the maximum length of the generated test cases as we restricted the ioco check by this limit.

We analysed the sets of conforming and non-conforming mutants produced by the ioco check and the combined conformance check. The combined approach wrongly classifies 14 mutants as conforming. However, all of them are covered by other test cases that were generated by our combined conformance check. In turn, the ioco check missed 223 mutants due to the smaller search depth. Note that these 223 mutants cannot be killed by the ioco test suite. We can quantify this difference in terms of the mutation score. In our setting the mutation score can be defined as the ratio of killed mutants to the total number of non-conforming mutants. The total number of non-conforming mutants detected by our checks is 942, whereby 928 have been found by the combined approach and additional 14 have been found by the ioco check. The test cases generated by the combined approach achieve a mutation score of 100%, whereas the stand-alone ioco check results in a test suite with a mutation score of $(942 - 223)/942 = 76\%$. Hence, our combined approach achieves a much higher mutation score on the model mutants than the ioco check.

Although the search depth was very limited compared to our combined approach, the ioco check takes considerably more time. While our combined approach took approximately 16 hours, the ioco check required 33.3 hours as can be seen from Table 6. Thereof, 22.8 hours were spent on checking 39% of the model mutants, which are the conforming mutants. On average, a non-conforming mutant requires 0.8 minutes, while a conforming mutant amounts to 2.9 minutes. The time required for writing logs is a negligible fraction compared to the overall run-time. Therefore, we did not state computation times without log file writing any more.

Table 6
Test case generation via ioco checking up to depth 10 for the PC_UML model. All values are given in minutes unless otherwise noted.

		not ioco	ioco	total
mutants [#]		719	466	1185
time – ioco check	Σ	9.8 h	22.8 h	32.6 h
	ϕ	0.8	2.9	1.7
	max	3.9	5.2	5.2
time – tc constr.	Σ	19	–	19
	ϕ	1.6 sec	–	1 sec
	max	5.8 sec	–	5.8 sec
total computation time	Σ	10.3 h	23 h	33.3 h
	ϕ	0.9	3	1.7
	max	3.9	5.2	5.2

6. Discussion

One advantage of our combined conformance check is that the refinement check avoids the costly ioco checking for many cases of conforming models. For the CAS_UML model the total effect is small, because it only has a total of 17 conforming mutants. However, 13 out of 17 ioco checks (76%) could be saved with the prior refinement checks (see Fig. 8). For the PC_UML model with the high number of 257 conforming mutants, we could save 189 out of 257 ioco checks (74%) with the prior refinement check (see Fig. 11).

However, as pointed out, our combined conformance check possibly misses some test cases as it might wrongly classify certain mutants as conforming, although they are not ioco-conform to the original model. Hence, it does not create a test case for these mutants and the generated test suite is an under-approximation of a full ioco test suite. This was not the case for any of the CAS_UML model mutants, while for the PC_UML model the percentage of wrongly classified mutants was only 1.2%. Note that for the particle counter model, this is a lower bound as the ioco checks were performed with lower exploration depth and hence classified some mutants as conforming, although they are possibly non-conforming in higher depths. However, it seems that the action systems generated from UML models have a beneficial structure that supports our approach. We investigated the CAS_UML model and found a plausible explanation: many intermediate results are saved in state variables and most mutations cause different states, which are detected by our refinement relation. Anyway, our under-approximation did in effect not lower the fault coverage of the generated test suites. All of the missed PC_UML mutants were covered by other test cases generated by our combined conformance check. Hence, the test cases killed all non-conforming mutants which means that the actual mutation score is 100%.

Furthermore, the assumption that is underlying our idea of combining refinement and ioco holds for our use cases: non-refinement propagates to an ioco violation in most cases. For the CS_UML model only 2.7% of the non-refining mutants were ioco conform and 6.8% in the case of the PC_UML model. That means that only a very small number of internal model faults did not propagate to an unexpected observable event. As can be seen from Figs. 9 and 12, the propagation only requires up to 5 steps for the majority of the cases.

Finally, our experiments demonstrated that our combination of refinement and ioco is important for complex models. For the simple CAS, our combined refinement and ioco check was only slightly faster than a pure ioco check. However, for the particle counter models, it turned out that our combined approach is essential in order to cope with such complex models. Our combined approach reduces the computation time by more than 50% for the PC_UML model.

Note that for the particle counter, the test suite generated by our combined approach turned out to be even stronger than those produced by the ioco checker. They achieve a higher mutation score (100% vs. 76%) on the given set of model mutants. The reason is that the exploration limits for the ioco checks needed to be smaller.

Threats to validity. Can these results be generalised? Two case studies are not enough to empirically support the claim that the combined approach is generally faster. However, the case studies highlight certain aspects that strongly indicate that this is the case. (1) Ioco checking is more complex than refinement checking. It has a game character and involves the synchronised exploration of two models, whereas the refinement check can be reduced to a reachability problem of finding an unsafe state in the original model. (2) The refinement check of conforming mutants saves many expensive ioco checks. All mutants with strengthened guards are in this class. (3) We assume that in most cases the path to a fault triggering event is long and therefore the refinement check pays off. This might not be true in all cases, but all interesting faults we want to cover are of this nature — they are deeply buried in the system's business logic. Simpler faults are easily detected via random testing.

7. Related research

Model-based mutation testing was first used for predicate-calculus specifications [20]. Later, Stocks applied mutation testing to formal Z specifications [21] 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 [22]. In contrast to this state-based equivalence test, our technique checks for refinement and input–output conformance. Thus, we support non-deterministic models.

In order to cope with non-determinism, Okun et al. suggest to synchronise non-deterministic choices in the original and the mutated model via common variables to avoid false positive counterexamples [23]. Boroday et al. propose two approaches that cope with non-determinism: modular model checking of a composition of the mutant and the specification, and incremental test generation via observers and traditional model checking [24].

The idea of using an ioco checker for mutation-based test-case generation comes from Weiglhofer, who tested a SIP registrar against mutated Lotos specifications [25]. Later, in order to support test models in the form of UML state machines and action systems, the new ioco checker *Ulysses* was developed [26,27,9]. It is an explicit ioco checker that supports a rich set of data types as well as hybrid system models, but shows low performance for highly parameterized actions. Therefore, the presented combined approach relying on an SMT solver was developed.

Sampaio et al. developed an alternative ioco checker for the process algebra CSP based on the model checker FDR [28]. They report that a comparison to Weiglhofer's explicit on-the-fly algorithm [25] showed that the FDR approach is about five times slower. However, their main contribution is the mechanical verification of the conformance checking algorithms in a theorem prover.

Recently, Noroozi et al. considered the simpler case of checking ioco for deterministic models [19]. They present a polynomial-time algorithm based on the reduction of checking ioco into the NHORNSAT problem. This is not surprising as it is known that for deterministic (input-enabled) models ioco coincides with alternating simulation [29] and that alternating simulation can be checked in polynomial time [30].

8. Conclusions

We have presented a novel conformance check for fault-based test case generation. First, a refinement check searches for faults, then an input–output conformance check explores if the detected fault propagates to a visible failure.

The refinement check is highly optimised and relies on symbolic techniques, like SMT solving. Therefore, it is fast in detecting faults, especially if they are hidden deep in the model. The assumption is that the fault propagation phase is usually short and, therefore, the more costly ioco check can be applied.

From the two experiments, we conclude that our combination is a valuable alternative to a stand-alone ioco check. We admit that for small systems like a car alarm system, it is not beneficial as a full ioco check is feasible. However, for more complex systems like the particle counter, it can drastically decrease the computation time, while at the same time the exploration depths and hence the fault coverage of the mutated models can be increased. For the more complex model of a measurement device, the computation time was decreased by more than 50%, while 223 more mutated models were detected.

To our knowledge this is the first conformance checking approach that combines refinement and ioco.

More recently, we have also optimised the tool with respect to avoiding the generation of duplicate test cases. In the most recent version, the tool first checks if any existing test case already kills a given mutant before a new test case is generated. Most recently, we have re-implemented an optimised version of the ioco check. It relies on symbolic execution and the first experiments are very promising. Furthermore, we investigate statistical model checking techniques to support stochastic fault models. This would lead to a probabilistic notion of risk, like in Murta and Oliveira's work [1].

Acknowledgement

We thank the three anonymous reviewers for their detailed and constructive feedback that helped to improve this article considerably. Research herein was 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).

References

[1] D. Murta, J.N. Oliveira, A study of risk-aware program transformation, Sci. Comput. Program. 110 (2015) 51–77.
[2] B. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, S. Tiran, MoMuT::UML – model-based mutation testing for UML, in: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13–17, 2015, IEEE, 2015, pp. 1–8.
[3] S. Tiran, On the effects of UML modeling styles in model-based mutation testing, Master's thesis, Graz University of Technology, Institute for Software Technology, May 2013.
[4] I.E.E.E. Computer, Society, IEEE standard classification for software anomalies, Tech. Rep. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), IEEE, January 2010.
[5] J. Tretmans, Test generation with inputs, outputs and repetitive quiescence, Softw., Concepts Tools 17 (3) (1996) 103–120.
[6] B.K. Aichernig, J. Auer, E. Jöbstl, R. Korošec, W. Krenn, R. Schlick, B.V. Schmidt, Model-based mutation testing of an industrial measurement device, in: Proceedings of Tests and Proofs – 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24–25, 2014, in: Lecture Notes in Computer Science, vol. 8570, Springer-Verlag, 2014, pp. 1–9.
[7] B.K. Aichernig, E. Jöbstl, S. Tiran, Model-based mutation testing via symbolic refinement checking, Sci. Comput. Program. 97 (2015) 383–404.
[8] H. Brandl, M. Weiglhofer, B.K. Aichernig, Automated conformance verification of hybrid systems, in: J. Wang, W.K. Chan, F.-C. Kuo (Eds.), Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14–15 July 2010, IEEE Computer Society, 2010, pp. 3–12.

[9] B.K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, S. Tiran, Killing strategies for model-based mutation testing, Softw. Test. Verif. Reliab. 25 (2014) 716–748.
[10] E. Jöbstl, Model-based mutation testing with constraint and SMT solvers, Ph.D. thesis, Institute for Software Technology, Graz University of Technology, April 2014.
[11] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: help for the practicing programmer, IEEE Comput. 11 (4) (1978) 34–41.
[12] W. Krenn, R. Schlick, B.K. Aichernig, Mapping UML to labeled transition systems for test-case generation – a translation via object-oriented action systems, in: FMCO, in: Lecture Notes in Computer Science, vol. 6286, Springer, 2010, pp. 186–207.
[13] G. Nelson, A generalization of Dijkstra's calculus, ACM Trans. Program. Lang. Syst. 11 (4) (1989) 517–561.
[14] R.-J. Back, R. Kurki-Suonio, Decentralization of process nets with centralized control, in: PODC, ACM, 1983, pp. 131–142.
[15] B.K. Aichernig, Mutation testing in the refinement calculus, Form. Asp. Comput. 15 (2–3) (2003) 280–295.
[16] B.K. Aichernig, J. He, Mutation testing in UTP, Form. Asp. Comput. 21 (1–2) (2009) 33–64.
[17] C. Hoare, J. He, Unifying Theories of Programming, Prentice-Hall, 1998.
[18] M. Hermann, Constrained reachability is NP-complete, short note, URL http://www.lix.polytechnique.fr/~hermann/publications/const-reach.pdf, 1998.
[19] N. Noroozi, M.R. Mousavi, T.A.C. Willemse, On the complexity of input output conformance testing, in: Formal Aspects of Component Software: 10th International Symposium, FACS 2013, Nanchang, China, October 27–29, 2013, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 8348, Springer, 2014, pp. 291–309.
[20] T.A. Budd, A.S. Gopal, Program testing by specification mutation, Comput. Lang. 10 (1) (1985) 63–73.
[21] P.A. Stocks, Applying formal methods to software testing, Ph.D. thesis, Department of Computer Science, University of Queensland, 1993.
[22] P.E. Ammann, P.E. Black, W. Majurski, Using model checking to generate tests from specifications, in: 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), IEEE, 1998, pp. 46–54.
[23] V. Okun, P.E. Black, Y. Yesha, Testing with model checker: insuring fault visibility, in: Proceedings of the International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems, 2003, pp. 1351–1356.
[24] S. Boroday, A. Petrenko, R. Groz, Can a model checker generate tests for non-deterministic systems? Electron. Notes Theor. Comput. Sci. 190 (2) (2007) 3–19.
[25] B.K. Aichernig, B. Peischl, M. Weiglhofer, F. Wotawa, Protocol conformance testing a SIP registrar: an industrial application of formal methods, in: 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), IEEE, 2007, pp. 215–224.
[26] B.K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, Model-based mutation testing of hybrid systems, in: Formal Methods for Components and Objects (FMCO) 2009, in: Lecture Notes in Computer Science, vol. 6286, Springer, 2010, pp. 228–249.
[27] H. Brandl, M. Weiglhofer, B.K. Aichernig, Automated conformance verification of hybrid systems, in: International Conference on Quality Software, IEEE, 2010, pp. 3–12.
[28] A. Sampaio, S. Nogueira, A. Mota, Y. Isobe, Sound and mechanised compositional verification of input–output conformance, Softw. Test. Verif. Reliab. 24 (4) (2014) 289–319.
[29] M. Veanes, N. Bjørner, Alternating simulation and IOCO, in: Proceedings of the 22nd International Conference on Testing Software and Systems (ICTSS 2010), in: LNCS, vol. 6435, Springer, 2010, pp. 47–62.
[30] K. Chatterjee, S. Chaubal, P. Kamath, Faster algorithms for alternating refinement relations, in: Computer Science Logic (CSL'12), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 16, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 167–182.