# Formal Test-Driven Development with Verified Test Cases

Bernhard K. Aichernig[1], Florian Lorber[1] and Stefan Tiran[1,2]
[1]*Institute for Software Technology, Graz University of Technology, Graz, Austria*
[2]*Safety & Security, Austrian Institute of Technology, Vienna, Austria*

Keywords: Model-based Testing, Formal Models, Model Checking, Test-Driven Development.

Abstract: In this paper we propose the combination of several techniques into an agile formal development process: model-based testing, formal models, refinement of models, model checking, and test-driven development. The motivation is a smooth integration of formal techniques into an existing development cycle. Formal models are used to generate abstract test cases. These abstract tests are verified against requirement properties by means of model checking. The motivation for verifying the tests and not the model is two-fold: (1) in a typical safety-certification process the test cases are essential, not the models, (2) many common modelling tools do not provide a model checker. We refine the models, check refinement, and generate additional test cases capturing the newly added details. The final refinement step from a model to code is done with classical test-driven development. Hence, a developer implements one generated and formally verified test case after another, until all tests pass. The process is scalable to actual needs. Emphasis can be shifted between formal refinement of models and test-driven development. A car alarm system serves as a demonstrating case-study. We use Back's Action Systems as modelling language and mutation analysis for test case generation. We define refinement as input-output conformance (ioco). Model checking is done with the CADP toolbox.

## 1 INTRODUCTION

Test-Driven Development (TDD) (Beck, 2003) places test cases at the centre of the development process. The test cases serve as specification, hence they have to be written before implementing the functionality. Furthermore, the functionality is only gradually increased, implementing test case after test case. The proposed development cycle is (1) write a test case that fails, (2) write code such that the test case passes, (3) refactor the code and continue with Step 1. It has been reported that TDD is able to reduce the defect rate by 50% (Maximilien and Williams, 2003).

However, experience over time shows that these test cases become part of the code-base and need to be maintained and refactored as well. "With time TDD tests will be duplicated which will make management and currency of tests more difficult (as functionality changes or evolves). Taking time to refactor tests is a good investment that can help alleviate future releases development frustrations and improve the TDD test bank." (Sanchez et al., 2007) Therefore, we propose the use of abstract test cases, that are less exposed to changes in the interface. Only the test adaptor mapping the abstract test cases to the (current) interface should be changed.

However, when functionality changes even the abstract test cases need updates. Instead of manually editing hundreds of test cases, we propose their automatic regeneration from updated models. This is what model-based testing adds to our process (Utting and Legeard, 2007). Hence, we propose to keep the test-driven programming style, but use abstract test models and model-based test-case generation to overcome the challenge of maintaining large sets of concrete unit-tests. Models are in general more stable to changes than implementations. Furthermore, the abstract models of the system under test (SUT) serve as oracles, specifying the expected observations for given input stimuli.

When using models for generating test cases, the models are critical. In model-based testing, wrong models lead to wrong test cases. Therefore, it is important to validate and/or verify the models against the expected properties. A prerequisite for this are formal models with a precise semantics. We use Back's Action Systems (Back and Kurki-Suonio, 1983) for modelling embedded systems under test.

For industrial users a translator from UML state machines to Action Systems exists (Krenn et al.,


Aichernig B., Lorber F. and Tiran S..
Formal Test-Driven Development with Verified Test Cases.
DOI: 10.5220/0004874406260635
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MBAT-2014), pages 626-635
ISBN: 978-989-758-007-9

2009), but in this work we will limit ourselves to Action Systems. The Action Systems are extended with parametrised labels and interpreted as labelled transition systems (LTS). For testing, the labels are partitioned into controllable (input), observable (output) and internal actions.

Since most model-based testing tools do not provide a model checker, we propose a pragmatic approach for checking the validity of models: we generate a representative set of test cases from the model, import these test cases as a model into a model checker in which we verify the required properties of the SUT. For example, we check if a given safety property is satisfied in the generated test cases. If this is not the case, this reflects a problem with the model, assuming that the test case generation works correctly. This approach places the test cases in the centre of formal development. The models are a means for test case generation.

As a further formal technique, we propose refinement techniques to develop a series of partial models into a refined more detailed model. In this work our notion of refinement is Tretmans' (Tretmans, 1996) input-output conformance relation (ioco). The partial models shall capture different functional aspects of a system, contributing to test cases focusing on the different functionality. More refined models, will generate test cases focusing on more subtle behaviour. A refined model may combine the partial models into a single model and possibly add new behaviour. The conformance relation ioco supports this kind of refinement (in contrast to, e.g., trace inclusion).

Our testing technique is regression based, in the sense that we only generate tests for new aspects in the refined models. Via automated refinement checking we ensure that the original properties of the abstract models are preserved.

Implementation follows an agile style via test-driven development. As soon as the first tests from the small partial models have been generated and verified in the model checker, the developers implement test after test incrementally.

This approach is based on the tool Ulysses (Aichernig et al., 2011; Aichernig et al., 2010; Brandl et al., 2010). Ulysses is an input-output conformance checker for Action Systems. In case of non-conformance between two models, a test case is generated that shows their different behaviour. Ulysses has been designed to support model-based mutation testing. In this scenario, we generate a number of faulty models from an original reference model. The faulty models are called mutants. Then, Ulysses checks the conformance between the original and the mutants producing test cases as counter-examples for conformance. We say that these tests cover the injected faults. Hence, our coverage criterion is fault-based. The generated test cases are then executed on the SUT. This method represents a generalisation of the classical mutation testing technique (Hamlet, 1977; DeMillo et al., 1978; Jia and Harman, 2011) to model-based testing. The testing technique has been presented before (Aichernig et al., 2011), here we extend it to a formal test-driven development process.

Note that Ulysses allows non-deterministic models, in which case adaptive test cases are generated. An adaptive test case has a tree-like shape, branching when several possible observations for one stimulus are possible.

We see our contributions as follows: (1) a new formal test-driven development process that is agile, (2) the idea to verify model-based test cases if direct verification of the model is impossible, (3) a new test case generation technique with mutation analysis under step-wise refinement of test models.

**Structure.** In the following Section 2 we present our case-study of a car alarm system. It will serve as a running example. Then, in Section 3 we give a detailed overview of our formal test-driven development process. The empirical results of our case study are presented and discussed in Section 4. Finally, we draw our conclusions in Section 5.

## 2 RUNNING EXAMPLE: A CAR ALARM SYSTEM

A car alarm system (CAS) serves as our running example. The example is inspired from Ford's automotive demonstrator within the past EU FP7 project MOGENTES[1]. The list of user requirements for the CAS is short:

**Requirement 1: Arming.** The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment and all doors are closed.

**Requirement 2: Alarm.** The alarm sounds for 30 seconds if an unauthorised person opens the door, the luggage compartment or the bonnet. The hazard flasher lights will flash for five minutes.

**Requirement 3: Deactivation.** The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

The system is highly underspecified and a variety of design decisions have to be considered. Furthermore,

[1] http://www.mogentes.eu

the timing requirements make the example interesting.

In the following section, we give an overview of our development process and refer to the example where appropriate.

# 3 A FORMAL TEST-DRIVEN DEVELOPMENT PROCESS

## 3.1 Initial Phase

At the beginning of the test-driven development process, we fix the testing interface of the system under development. First, the system boundaries are determined. It must be clear what kind of functionality belongs to the system under test and what belongs to the environment. At the most abstract level, we enumerate the controllable and observable events. The controllable actions represent stimuli to the system. The observable actions are reactions from the system and are received by the environment. The test driver will emulate this environment. The abstract test cases are expressed in terms of these abstract controllable and observable actions. Actions can have parameters.

**Example 3.1.** For the CAS we identified the following controllable events: *Close*, *Open* for closing and opening the doors, and *Lock*, and *Unlock* for locking and unlocking the car.

The observables are *ArmedOn*, *ArmedOff* for signalling that the CAS is arming and disarming. In the real car, a red LED will blink to signal the armed state. Furthermore, we can observe the triggering of the sound alarm, *SoundOn*, *SoundOff*, and flash alarm, *FlashOn*, *FlashOff*.

In addition, we parametrise each event by time. For a controllable event $c(t)$ this means that the event should be initiated by the tester after $t$ seconds since the last event. An observable event $o(t)$ must occur after $t$ seconds. These hard deadlines can be relaxed in the test driver. For example, the observable *ArmedOn(20)* denotes our expectation that the CAS will switch to armed after 20 seconds. □

Second, the test interface at the implementation level has to be fixed. For each abstract controllable event, a concrete stimulus has to be provided by the system under test. In addition, the test driver has to provide an interface for the actual observations.

Next, the test driver is implemented. It takes an abstract test case as input and runs the tests as follows: (1) the abstract controllables are mapped to concrete input stimuli, (2) the concrete input is executed on the system under test, (3) the actual output is mapped back to the abstract level and compared to the expected outputs. This is classical *offline testing* (Utting et al., 2011). Note that we allow nondeterminism in both the models and the systems under test. Hence, our test cases may be adaptive, i.e. they may have a tree-like shape (Hierons, 2006). The test driver may issue three different verdicts: *pass*, *fail* and *inconclusive*. The latter is used to stop an adaptive test run, if a given test purpose cannot be met.

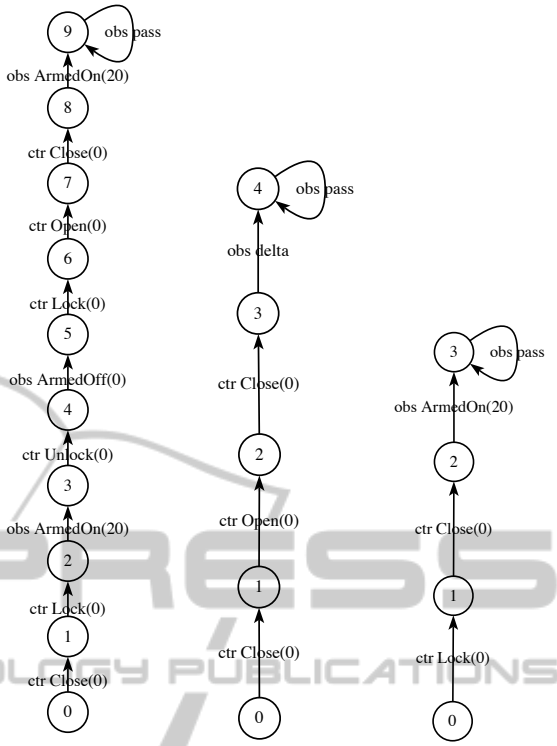


Figure 1: Three abstract test cases for the CAS based on the first partial model.

The abstract test cases are represented in the Aldebaran format of the CADP toolbox[2] . This is a simple graph notation for labelled transition systems.

**Example 3.2.** For our experiment we implement the CAS in Java. Hence, the test driver maps controllable events to method calls to the SUT. The observables are realised via callbacks to the test driver. With respect to timing we opted for simulated time. The test driver sends tick events to the SUT, signaling the elapse of one second.

Figure 1 shows three abstract test cases that can be executed by the test driver. This graph representation of the test cases is automatically drawn with CADP. □

Here, we want to point out the advantage of splitting the test cases into abstract test cases and a test

[2]http://www.inrialpes.fr/vasy/cadp/

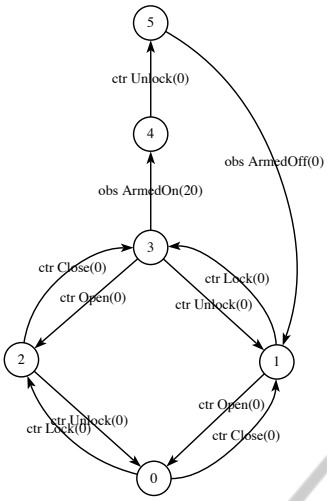


Figure 2: The labelled transition system semantics of the partial model of Figure 4.

driver. When we change the interface of an implementation, only the test driver needs to be updated. For example, when changing the CAS implementation from simulated time to real-time, the abstract test cases remain valid. In contrast, any concrete JUnit test case with timing properties would need adaptation.

After this initial phase, we enter the iterative phase of our agile process. The cycle is shown in Figure 3. First, a small partial model is created. This model captures some basic functionality that should be implemented first. Then, a set of abstract test cases is generated from the model. Next, we verify the test cases with respect to temporal properties. Then, the test cases are implemented in a test-driven fashion. Finally, the implementation is refactored. After this cycle, a different aspect of the system is modelled or a given model is refined and the cycle starts again. In the following, we present the techniques supporting this process.

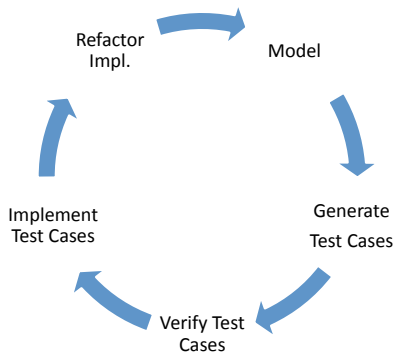


Figure 3: The formal test-driven development cycle.

## 3.2 Model

The used formal modelling notation is a version of Object-Oriented Action Systems (OOAS) (Bonsangue et al., 1998), an object-oriented extension of Back's classical Action Systems (Back and Kurki-Suonio, 1983). However, in this paper we will not make use of the object-oriented features.

A classical Action System consists of a set of variables, an initial state and one loop over a non-deterministic choice of actions. An action is a guarded command that updates the state, if it is enabled. The action system iterates as long as a guard is enabled and terminates otherwise. In case of several enabled guards, one is chosen non-deterministically.

In the used version of Action Systems, actions may be sequentially or non-deterministically composed as well as nested. For testing, each action has been instrumented with a parametrised label. These labels represent the abstract events. Hence, actions are categorised into controllable and observable events. In addition, internal actions are allowed.

**Example 3.3.** Figure 4 shows a partial Action System model of the CAS. Its purpose is to express the arming and disarming behaviour. After two basic type definitions (Line 2-3) the Action System class is defined. Its initialisation is expressed in the system block at the bottom (Line 36). Here, several objects could be assembled via composition operators. Three Boolean variables define the state space of the Action System (Line 8-9). Next, we present three actions (Line 11-25). The controllable action $Close(t)$ is only enabled in the state "disarmed and doors open". Furthermore, it must happen immediately (Line 13). The action sets the variable closed to true. Similarly the controllable $Lock(t)$ is defined (Line 16-19). The observable $ArmedOn(t)$ happens after 20 seconds, if the car is locked and doors are closed (Line 21-25). The remaining three actions follow the same style and are omitted for brevity. Finally, in the do-od block (Line 28-33) the actions are composed via non-deterministic choice. □

The operational semantics of an action system is defined by a labelled transition system (LTS) with transition relation $T$ as follows: if in a given state $s$ an action with label $l$ is enabled, resulting in a post state $s'$, then $(s,l,s') \in T$.

**Example 3.4.** Figure 2 shows the LTS of the Action System of Figure 4. □

## 3.3 Generate Test Cases

In this phase a model-based testing tool generates the test cases from the partial models. Different strate-

```
types
  TimeSteps = {I0=0 , I20=20, I30=30, I270=270};
  Int = int[0..270];
  AlarmSystem = autocons system
  |[
    var
      closed: bool = false ;
      locked: bool = false ;
      armed: bool = false
    actions
      ctr Close( waittime : Int) =
        requires not armed and not closed and
            waittime = 0:
          closed := true
        end ;
      ctr Lock( waittime : Int) =
        requires not armed and not locked and
            waittime = 0:
          locked := true
        end ;
      obs ArmedOn(waittime : Int) =
        requires (waittime = 20 and not armed and
            locked and closed):
          armed := true
        end ;
      ...
  do
    var t : TimeSteps : Close(t)   []
    var t : TimeSteps : Open(t)    []
    var t : TimeSteps : Lock(t)    []
    var t : TimeSteps : Unlock(t)  []
    var t : TimeSteps : ArmedOn(t) []
    var t : TimeSteps : ArmedOff(t)
  od
  ]|
system AlarmSystem
```

Figure 4: Partial model describing the arming of the CAS.

gies for generating the test cases from models exist. We use the test case generator Ulysses which supports random test case gneration and model-based mutation testing from Action Systems (Brandl et al., 2010; Aichernig et al., 2011).

Random generation produces long unbiased test cases but has no stopping criterion. Mutation testing adds a fault-centred approach. The goal is to cover as many possible faults as anticipated in the model. In the following we concentrate on the mutation testing approach.

Ulysses is realised as a conformance checker for Action Systems. It takes two Action Systems, an original and a mutated one, and generates a test case that kills the mutant. Killing a mutant means that the test case can distinguish the original from the mutated model. The mutated models, i.e. the mutants, are automatically generated by so-called mutation operators. They inject faults into a given model, in our case one fault per mutant.

Ulysses expects the actions being labelled as controllable, observable and internal actions. For deterministic models, the generated test case is a sequence of events leading to the faulty behaviour in the mutant. For non-deterministic models an adaptive test case with inclonclusive verdicts is generated. Ulysses explores both labelled Action Systems, determinizes them, and produces a synchronous product modulo the ioco conformance relation of Tretmans (Tretmans, 1996). The ioco relation supports non-deterministic, partial models. Ulysses is implemented in Sicstus Prolog exploiting the backtracking facilities during the model explorations.

Different strategies for selecting the test cases from this product are supported: linear test cases to each fault, adaptive test cases to each fault, adaptive test cases to one fault. Ulysses also checks if a given or previously generated test case is able to kill a mutant. Only if none of the test cases in a directory can kill a new mutant, a new test case is generated. Furthermore, as mentioned, Ulysses is able to generate test cases randomly. Our experiments showed that for complex models it is beneficial to generate first a number of long random tests for killing the most trivial mutants. Only when the randomly generated tests cannot kill a mutant, the computationally more expensive product calculation is started. The different strategies for generating test cases are reported in (Aichernig et al., 2011).

**Example 3.5.** Our mutation tool produces 114 mutants out of the partial model of Figure 4. Ulysses generates 12 linear test cases killing all of these mutants. These test cases ensure that none of the 114 faulty versions will be implemented. □

Next, we discuss how we run additional verification on the test cases before implementing them.

## 3.4 Verify Test Cases

In this phase, we verify different temporal properties of the generated test cases. This ensures that the generated test cases satisfy our original requirements. If wrong tests have been generated due to modelling errors, this would be detected. This provides the necessary trust in the test cases required for safety certification. The advantage of this method is that a special-purpose test case generator and a model checker can be combined without integrating them into a tool set. The mapping from abstract test cases into a model checker language is trivial.

We use the model checker of CADP, a toolbox for the design of communication protocols and distributed systems. It offers a wide set of functionality

for the analysis of labelled transition systems, ranging from step-by-step simulation to massively parallel model-checking.

Ulysses is able to generate the test cases as labelled transition systems in one of CADP's input formats, namely the Aldebaran format. These are text files describing the vertices and edges in a labelled directed graph. These test cases in could already be processed by the model checker without conversion. For checking safety properties this would already be sufficient. However, if we merge all test cases, we can additionally define properties to ensure that certain scenarios are covered by the test suite. In the following, we discuss how we merge the set of generated test cases into a single model.

**Merging of Test Cases.** The merging of the test cases into a model for analysis comprises three steps.

First, we copy all test cases in one single file and rename the vertices in order to keep unique identifiers. The only exception are the start vertices that share the same identifier. This joins all test cases in the start state. After this syntactic joining the file is converted into a more efficient binary representation (BCG format).

Second, we use the CADP Reductor tool to simplify the joint test cases via non-deterministic automata determinisation. This determinisation follows a classical subset construction and is initiated with the *traces* option. The determinisation merges the common prefixes of test cases.

Third, the CADP Reductor tool is applied again. This time we run a simplification that merges states that are strongly bisimilar (option *strong*)[3].

**Example 3.6.** Figure 5 shows the 12 merged test cases of Figure 4. The common end state of this model is the pass state of all test cases. □

This simplification is actually not necessary for the following verification process. However, the elimination of redundant parts facilitates the visual inspection of the behaviour defined by the test cases. Furthermore, we observed that the visualisation of the simplified model provides an insight into the redundancy of the test cases: the simpler the resulting model, the more redundant were the original test cases.

**Verification of Test Cases.** CADP provides the Evaluator tool, an on-the-fly model checker for labelled transition systems. Evaluator expects temporal properties expressed as regular alternation-free mu-calculus formula (Mateescu and Sighireanu, 2003). It

[3]Note that our test cases have no internal transitions, hence, strong and weak bisimulation are equivalent.

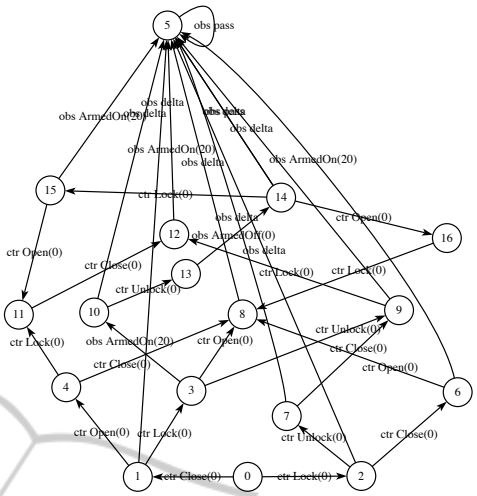


Figure 5: The LTS after merging the 12 test cases generated from Figure 4.

is an extension of the alternation-free fragment of the modal mu-calculus with action predicates and regular expressions over action sequences.

**Example 3.7.** We checked several safety properties related to our requirements. For example, the following temporal Property P1 is satisfied by our test cases.

```
[(not 'ctr Lock.*')* . 'ctr Close.*' .          (P1)
(not 'ctr Open.*')*  . 'ctr Lock.*' .
(not ('obs ArmedOn(20)' or 'ctr .*(.)' or
'ctr .*(1.)' or 'obs pass'))] false
```

It partly formalises Requirement 1 and says that if the doors are closed and locked it must not happen (expressed by the `false` at the end) that any controllable with more than 20 seconds as parameter or any other observable than the activation of the alarm system occurs. Note that in mu-calculus the states are expressed via event histories. Here, the state closed and locked is expressed via a sequence of events: the doors had been first closed and not later opened etc. □

We can also check for test case completeness in the sense that we verify that certain traces are included in our test cases.

**Example 3.8.** For example the next Property P2 checks if a trace with first locking and then closing the doors leading to an armed state is included:

```
<true*><'ctr Lock.*'>                           (P2)
<(not 'ctr Unlock.*')*> <'ctr Close.*'>
<(not 'ctr Unlock.*' and not 'ctr Open.*')*>
<'obs ArmedOn(20)'> true
```

Here the diamond operator `<.>` is used to express the existence of traces. □

Our integration with the CADP toolbox also allows us to check if certain scenarios are included in a

test suite by checking if certain test purposes are covered by the test cases. A test purpose is a specification for test cases expressing a certain test goal.

**Example 3.9.** The next Property P3 was implemented to guarantee the inclusion of a scenario in which the alarm is turned off. Which of the two alarms is turned off first is underspecified. Hence, it allows both scenarios.

```
(<true*> <'obs SoundOff.*'>                    (P3)
<'obs FlashOff.*'> true) or
(<true*> <'obs FlashOff.*'>
<'obs SoundOff.*'> true)
```

□

### 3.5 Implementation and Refactoring

Once the test cases are formally checked, we implement them. Here, the partial models serve to group them into functional units. Alternatively, for larger models, the test purposes may serve to categorise the test cases. A negated test purpose property will report the test case to be implemented next. In general, the developer starts with the shortest test cases and adds functionality until all tests pass.

As in common test-driven development, after a (set of) tests pass, the implementation is refactored. Hence, the code is simplified and rechecked against the existing test cases.

**Next Iteration.** After this phase, the development cycle starts again with either (1) a further partial model, capturing a different aspect of the system, or (2) a refined model adding details to existing models. Our refinement relation is the input-output conformance (ioco) relation. Hence, we can check the refinement of our models with the ioco-checker Ulysses. In the following, we discuss the empirical results of this process for the CAS.

## 4 EMPIRICAL RESULTS

In the following, we report the results of developing the CAS over several development cycles. We implemented in Java. By definition of TDD all test cases pass our implementation. Therefore, in order to evaluate the quality of our generated test cases, we run a mutation analysis on the implementation level. For this we use 38 faulty Java implementations of the CAS already used in previous work (Aichernig et al., 2011). The novelty in this paper is that we analyse how the mutation score develops under refinement. The mutation score is the number of killed mutants divided by the total number of mutants. We eliminated equivalent mutants.

For test case generation, we used the strategy A5 reported in (Aichernig et al., 2011): With this lazy strategy, Ulysses first checks whether any of the previously created test cases is able to kill the mutated model before a new test case is generated. We have also taken the eight test purposes used in (Aichernig et al., 2011) and analysed at what refinement step they are satisfied by the generated test cases.

**Iteration 1.** The first iteration in our development process covers the partial model of Figure 4. The first line of Table 1 ($CAS_1$) summarises the results.

As already mentioned, we generated 114 model mutants from which Ulysses generated 12 test cases. These tests were able to kill 81% of the implementation mutants. The tests passed our 18 safety checks, but were neither complete nor did they cover all test purposes. This is obvious, since only a part of the functionality was captured in the model and our properties required full functionality.

**Iteration 2.** In the second iteration, the triggering of the sound and flash alarm was modelled. Figure 6 shows the LTS of this Action System model. This model is not input-output conform to $CAS_1$, since it includes only one trace arming the system. The second line of Table 1 shows the results of this iteration ($CAS_2$). This model produces a high number of mutants (1889), but they result in 17 test cases only. The reason is that we selected all mutation operators in our tool. The low number of test cases generated indicates that we could have done with a smaller subset. These 17 test cases kill 73% of the Java mutants. Note that this model already satisfies all test purposes.

The test cases of both models together ($CAS_{1+2}$) already kill 97% of all faulty implementations. Furthermore, all required completeness properties of the test case are satisfied.

**Iteration 3.** In the third iteration, we merged the behaviour of the first two iterations into one Action System model. Figure 7 presents the corresponding LTS semantics. We checked with Ulysses that this integrated model input-output conforms to the two previous ones. Hence, we formally verified that the new model is a refinement of both partial models. Note the different use of Ulysses. Previously, we used the conformance checker to generate test cases by comparing a model with mutated version. Here, we first checked a complete model against a partial model to show that we did not introduce unwanted behaviour.

Table 1: Quality check of the test cases over the iterations, measured in mutation score on the faulty implementations and by model checking of the merged test cases.

| | # Mutants | # Test cases | Mutation Score | Safety | Completeness | Purposes |
|---|---|---|---|---|---|---|
| $CAS_1$ | 114 | 12 | 81% | 18/18 | 10/25 | 3/8 |
| $CAS_2$ | 1889 | 17 | 73% | 18/18 | 24/25 | 8/8 |
| $CAS_{1+2}$ | 2003 | 29 | 97% | 18/18 | 25/25 | 8/8 |
| $CAS_3$ | 2179 | 53 | 100% | 18/18 | 25/25 | 8/8 |
| $CAS_{1+2+3}$ | 2179 | 54 | 100% | 18/18 | 25/25 | 8/8 |

The fourth line of Table 1 ($CAS_3$) shows that the integrated model adds value. The combined behaviour leads to 176 new mutants ($2179 - 2003$). The number of test cases has increased, too. The result is a mutation score of 100%. Hence, all of 38 faulty Java implementations have been detected with these 53 test cases. We argue that this maximal mutation score provides a high trustworthiness in our test suite. Hence, we implemented these test cases in a test-driven style and stopped the iterative development at this point.

The results of a further experiment are shown in the bottom line of Table 1 ($CAS_{1+2+3}$). Again, Ulysses takes the *CAS*$_3$ model and its 2179 mutants as input. The difference here is the test case generation. Ulysses starts with the given 29 test cases of the previous two iterations. Hence, for every mutant Ulysses first checks, if it can be killed by the given tests. This is a kind of regression test case generation under model refinement. The results are very similar, except that in total one more test case has been generated. The reason is that the test cases of Iteration 1 are very short and longer tests subsuming them are added during the process. Currently, we do not post process the test cases in order to minimise their number. Neither do we order the given test cases, which would be beneficial. This is future work.

## 4.1 Discussion

In the following, we discuss some of the pros and cons of this approach as experienced in the case study.

**Benefits.** The proposed development process combines the advantages of three disciplines: (1) model-based testing, (2) test-driven development, and (3) formal methods. Classical test-driven development is ad-hoc, in the sense that the implementation will be as good as the test designer: manually designed tests may be incorrect and/or incomplete. Consequently, the implementation may be incorrect and/or incomplete. Our approach guarantees a correct and complete test suite. Generating the test cases systematically from a model gives a certain kind of coverage. In our case it is a fault coverage. However, the model may be incorrect. Therefore, it has to be checked against the requirements. Our approach allows to do so even if the modelling tool does not support model checking. The importing of abstract test cases into a model checker is easy. This allows us to check certain safety invariants indirectly. We can also add manually designed abstract test cases, or combine test cases from different tools. The incompleteness issue is checked via completeness properties and test purposes. The latter links the test cases to the requirements, although the model does not refer to them. Traceability is an important aspect. Negating a test purpose property and checking it, will immediately report a test case that covers this test purpose.

The iterative process with refinement adds more and more functionality to the models. However, in contrast to classical refinement from an abstract model to the implementation code, we immediately start coding in the first iteration. This combines the advantages of a formal process with agile iterative

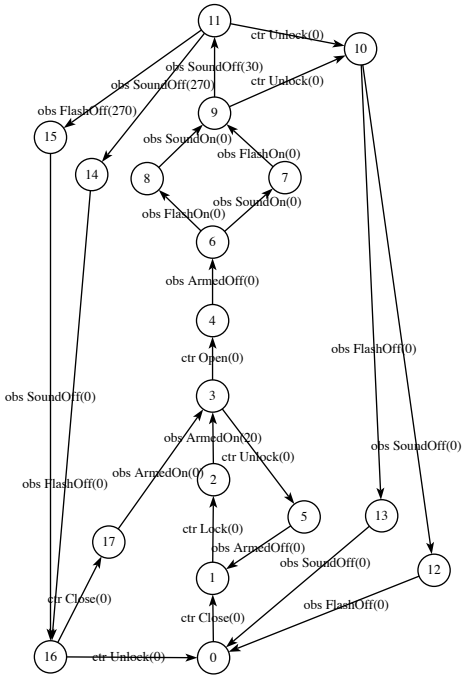


Figure 6: The second partial model of the CAS capturing only one trace to the armed state and all the traces afterwards.

methods.

The mutation analysis is not the main point of this paper, but it provided a coverage on the abstract modelling level as well as on the implementation level. We could show that the test cases covering all fault-models in the Action Systems, were sufficient to cover all faulty Java implementations. This adds a second, fault-centred, perspective to the completeness check of our test cases.

**Limitations.** It is obvious, that model checking test cases is not the same as verifying an implementation model. Many test cases may be required to capture the subtle cases of concurrent interleavings. Therefore, we have added the test purposes and completeness properties. In future, we may apply model-learning (Shahbaz and Groz, 2009) to merge the test cases into more concise models.

A further limitation concerns our mutation approach. Our new mutation tool for Action Systems produced far too many mutants. This leads to extremely long test case generation times. The 114 model mutants of $CAS_1$ could be processed in less than a minute. However, $CAS_2$ took almost 8 hours to process the 1889 mutants, and $CAS_3$ run 22 hours. In future, we will apply mutation avoidance techniques in order to reduce the number of mutants (Jia and Harman, 2011). Fortunately, the situation is not as severe as it might look like. The regression strategy of $CAS_{1+2+3}$ saved over 2 hours generation time. Furthermore, Ulysses continuously produces test cases while analysing the mutants. For example, after 10 seconds of analysing $CAS_2$ 4 test cases are available, after 1 minute 6, after 10 minutes 8, after 1 hour 14 of the 17 test case. Hence, one can perform the safety checks and implement the first test cases while Ulysses is looking for further test cases.

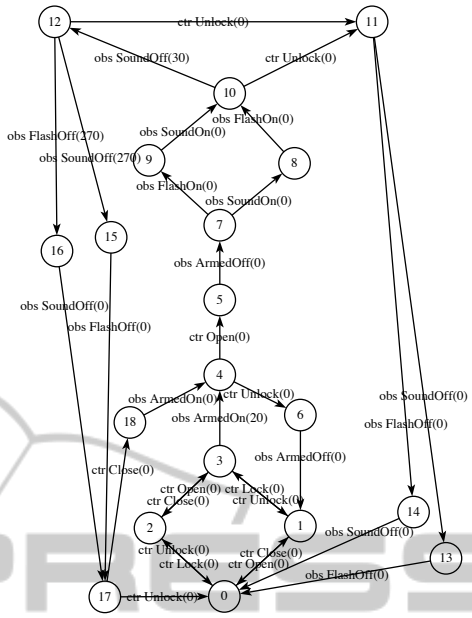


Figure 7: The third, complete model of the CAS containing all traces.

## 5 CONCLUSIONS

We have motivated and presented a formal test-driven development technique that combines the benefits of (1) model-based testing, (2) test-driven development, and (3) formal methods. The novelty of this approach is the model checking of the generated test cases as well as the combination of model refinement and test-driven development. In our experiments, we used mutation testing to generate the tests and to evaluate them at the implementation level. We have presented the first study of model-based mutation testing under model refinement. Our own tool Ulysses and the CADP toolbox automate the whole process.

Baumeister proposed the combination of TDD with formal specifications. His ideas differ to ours. In (Baumeister, 2004) he uses the tests to develop JML contracts. In (Baumeister et al., 2004) he proposes an iterative TDD process for developing UML state machines. His idea is to instrument the models with OCL constraints, but this was not implemented. It seems our approach is novel.

The used modelling notation is very similar to Event-B (Abrial, 2010). The reason is that Event-B was inspired by Action Systems. Event-B also identifies actions by labels. However, the refinement notions are different. We use input-output conformance defined over the input and output labels of the operational semantics, in contrast, Event-B applies the classical state-based notion of refinement via a weakest precondition semantics.

We are not the first who verified test cases. Niese et al. (Niese et al., 2001) model checked LTL properties of system-level test cases. In contrast to our work, these tests were not automatically generated from models, but designed with a domain-specific library. The verification of the test cases ensured that the test designer respected certain constraints, e.g. the existence of verdicts.

Of course, we are not the first who propose model-based mutation testing. A good survey can be found in (Jia and Harman, 2011). However, to the best of our knowledge this is the first experiment of applying it in combination with refinement of models.

In future, we will work on overcoming the discussed limitations and perform larger case studies.

## ACKNOWLEDGEMENTS

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement N° 269335 and from the Austrian Research Promotion Agency (FFG) under grant agreement N° 829817 for the implementation of the project MBAT, Combined Model-based Analysis and Testing of Embedded Systems.

## REFERENCES

Abrial, J.-R. (2010). *Modelling in Event-B: System and software design.* Cambridge University Press.

Aichernig, B. K., Brandl, H., Jöbstl, E., and Krenn, W. (2010). Model-based mutation testing of hybrid systems. In de Boer, F. S., Bonsangue, M. M., Hallerstede, S., and Leuschel, M., editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 228–249. Springer-Verlag.

Aichernig, B. K., Brandl, H., Jöbstl, E., and Krenn, W. (2011). Efficient mutation killers in action. In *IEEE Fourth International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21–25 , 2011*, pages 120–129. IEEE Computer Society.

Back, R.-J. and Kurki-Suonio, R. (1983). Decentralization of process nets with centralized control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142. ACM.

Baumeister, H. (2004). Combining formal specifications with test driven development. In *Extreme Programming and Agile Methods - XP/Agile Universe 2004, 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, August 15-18, 2004, Proceedings*, pages 1–12.

Baumeister, H., Knapp, A., and Wirsing, M. (2004). Property-driven development. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 96–102.

Beck, K. (2003). *Test Driven Development: By Example*. The Addison-Wesley Signature Series. Addison-Wesley.

Bonsangue, M. M., Kok, J. N., and Sere, K. (1998). An approach to object-orientation in action systems. In *Mathematics of Program Construction, LNCS 1422*, pages 68–95. Springer.

Brandl, H., Weiglhofer, M., and Aichernig, B. K. (2010). Automated conformance verification of hybrid systems. In *10th Int. Conf. on Quality Software (QSIC 2010)*, pages 3–12. IEEE Computer Society.

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

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

Hierons, R. M. (2006). Applying adaptive test cases to nondeterministic implementations. *Inf. Process. Lett.*, 98(2):56–60.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

Krenn, W., Schlick, R., and Aichernig, B. K. (2009). Mapping UML to labeled transition systems for test-case generation - a translation via object-oriented action systems. In *Formal Methods for Components and Objects (FMCO)*, pages 186–207.

Mateescu, R. and Sighireanu, M. (2003). Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255 – 281. Special issue on Formal Methods for Industrial Critical Systems.

Maximilien, E. and Williams, L. (2003). Assessing test-driven development at IBM. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 564 – 569.

Niese, O., Steffen, B., Margaria, T., Hagerer, A., Brune, G., and Ide, H.-D. (2001). Library-based design and consistency checking of system-level industrial test cases. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Genova, Italy, April 2-6, 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 233–248. Springer-Verlag.

Sanchez, J. C., Williams, L., and Maximilien, E. M. (2007). On the sustained use of a test-driven development practice at IBM. In *Proceedings of the AGILE 2007*, pages 5–14, Washington, DC, USA. IEEE Computer Society.

Shahbaz, M. and Groz, R. (2009). Inferring mealy machines. In *Proceedings of the 2nd World Congress on Formal Methods, FM'09*, LNCS, pages 207–222. Springer-Verlag.

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

Utting, M. and Legeard, B. (2007). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers.

Utting, M., Pretschner, A., and Legeard, B. (2011). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*.