Integrating Model-Based Testing and Analysis Tools via Test Case Exchange

Bernhard K. Aichernig, Florian Lorber, Stefan Tiran
Institute for Software Technology
Graz University of Technology, Austria
{*aichernig, florber, stiran*}*@ist.tugraz.at*

***Abstract*—Europe's industry in embedded system design is currently aiming for a better integration of tools that support their development, validation and verification processes. The idea is to combine model-driven development with model-based testing and model-based analysis. The interoperability of tools shall be achieved with the help of meta-models that facilitate the mapping between different modelling notations. However, the syntactic and semantic integration of tools is a complex and costly task. A common problem is that different tools support different subsets of a language. Furthermore, semantic differences are a major obstacle to sound integration efforts.**

In this paper we advocate an alternative, more pragmatic approach. We propose the exchange of test cases generated from the models instead of exchanging the models themselves. The advantage is that test cases have a much simpler syntax and semantics, and hence, the mapping between different tools is easier to implement and to maintain. With a formal testing approach with adequate testing criteria a set of test cases can be viewed as partial models that can be formally analysed. We demonstrate an integration of our test case generator Ulysses with the CADP toolbox by means of test case exchange. We generate test cases in Ulysses and verify properties in CADP. We also generate test cases in CADP and perform a mutation analysis in Ulysses.

***Keywords*- tool integration, model-based testing, mutation testing, model checking, Ulysses, CADP, TGV.**

I. Introduction

The area of high-quality embedded systems is an important sector for Europe's industry. Especially the development of highly-critical components for the transportation domain, i.e. avionics, railways, and automotive, is of strategic interest. In order to coordinate and foster the research and development in this area, the ARTEMIS Industry Association was founded in 2007. One of its strategic goals is to overcome the growing dependence on development tools in embedded systems design:

> "At present, large-scale development environments come almost exclusively from a small number of non-European sources, while Europe has a large number of excellent suppliers - mostly SMEs - of tools for specific purposes. This situation has created on the one hand a strong dependence on external suppliers for the necessary tool frameworks and on the other a highly fragmented supply chain within Europe for often critical, specialised development tools. Often the market for these tools is limited because they are not readily interoperable with existing frameworks." [4]

Therefore, ARTEMIS proposes to establish tool platforms with a common set of interfaces and protocols that will allow tool vendors to integrate their products. The topic of the ARTEMIS project CESAR (www.cesarproject.eu) is the integration of model-driven development tools. In the more recent ARTEMIS project MBAT (www.mbat-artemis.eu) this is extended to model-based testing and analysis tools.

The integration of model-based tools is a challenging task. Usually, not all tools support the full language and many have implemented syntactic and semantic variations of a common language standard — provided a common accepted standard exists. Furthermore, evolving domain specific extensions of a standard notation turn trustworthy interoperability into a running target.

One solution to tackle this challenge is to abstract from concrete syntax and define common meta-models, i.e. abstract syntax. In this case, the needed model transformations are defined over these meta-models. This approach is implemented, e.g. in the ModelBus framework of Fraunhofer [10].

In this paper we advocate an alternative, more pragmatic approach for integrating model-based test case generators and analysis tools: As an alternative to the exchange of models, we propose to exchange test cases that are automatically generated from the models.

The rationale behind this bold idea is that we can view a set of test cases, i.e. a test suite, as a partial model of a system under test. If we take the view that a test case defines one particular behaviour and leaves the rest undefined, it may serve as a specification. The more test cases we add to a test suite, the more refined our specification becomes. We took this specification view of test cases previously in our theoretical work on mutation testing [2]. Now, it forms the basis of a lightweight bridging between testing and analysis tools.

The test models in model-based testing are abstract and so are the generated test cases. These models shall capture the possible user interaction consisting of stimuli and expected reactions (observations). For test execution a test adaptor maps the abstract test cases to the concrete interfaces of the system-under-test. We propose to map these abstract test cases between different tools for analysis purposes. These abstract test cases have a simple syntax and semantics and

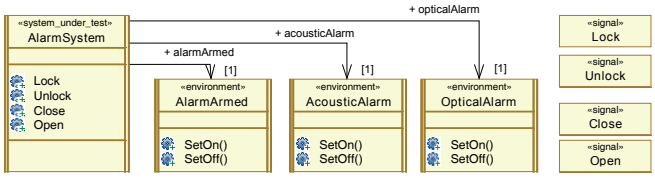


Figure 1. Car Alarm System - Testing Interface.

can therefore easily be mapped between different tools.

In this paper we demonstrate one such integration and application scenario. We generate test cases with our model-based test case generator Ulysses [7]. These test cases have the form of input-output sequences. We import these test cases into the CADP toolbox[1] and merge them into one model using the simplifiers of CADP. Then, we model check certain safety properties with the model checker of CADP. 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 tool chain works also in the opposite direction. CADP also provides a test case generator, called TGV [11], following a different testing strategy. We generate test cases in TGV and import them into Ulysses. Then, Ulysses performs a mutation analysis on the modelling level in order to assess the quality of these test cases with respect to fault-detection.

The rest of the paper is structured as follows: Section II presents our running example of a car alarm system. Next, Section III explains the test case generation technique implemented in Ulysses. Then, in Section IV we analyse these test cases with the model checker of CADP. Section V discusses test case generation in CADP, and Section VI presents their mutation analysis in Ulysses. Finally, we draw our conclusions in Section VII.

II. Running Example

A simplified car alarm system serves as our running example. The example is inspired from Ford's automotive demonstrator within the past EU FP7 project MOGENTES. The UML test model was created from the following list of requirements for the car alarm system (CAS):

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.

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

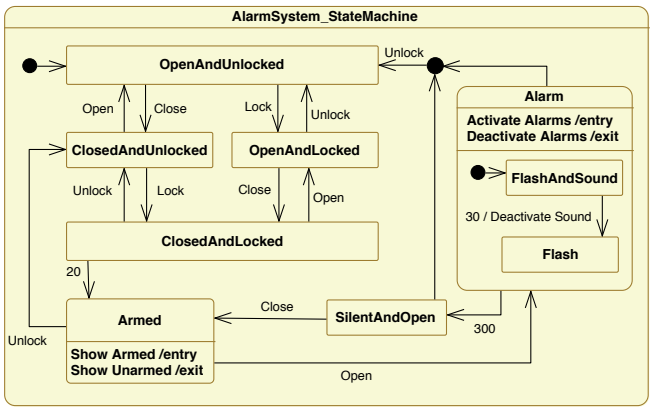


Figure 2. Car Alarm System - State Machine.

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.

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

A. Testing Interface

The UML model of the car alarm system comprises four classes and four signals, as shown in Figure 1. The class *AlarmSystem* is marked as system under test (SUT) and may receive any of the *Lock*, *Unlock*, *Close*, or *Open* signals. At the same time, the SUT calls methods of the classes *AlarmArmed*, *AcousticAlarm*, and *OpticalAlarm* – all of them marked as being part of the environment.

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

B. State Machine

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

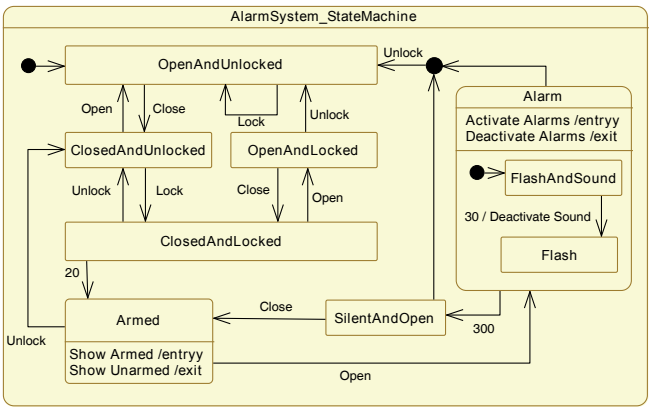


Figure 3. Car Alarm System - Mutated State Machine.

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

Notice that our semantics for UML-state machines differs slightly from the UML standard. In order to support partial test models, the state-machine only accepts events that trigger a transition. For example, the state-machine will only accept *Close* and *Lock* events when being in state *OpenAndUnlocked*.

III. Generating Tests in Ulysses

Ulysses is a test case generator following the model-based mutation testing strategy. This is similar to the mutation testing of programs. Program mutation testing provides a method of assessing and improving a test suite by checking if its test cases can detect a number of injected faults in a program. The faults are introduced by syntactically changing the source code following patterns of typical programming errors [8], [9]. For a good and recent survey on mutation testing see [12].

However, in our approach we apply model-based mutation testing. The idea is to mutate the UML models and generate those test cases that would kill a set of mutated models. The generated tests are then executed on the system under test (SUT) and will detect if a mutated UML state machine has been implemented. It is a complementary testing approach, well-suited for dependability analysis, since its coverage is measured in terms of faults.

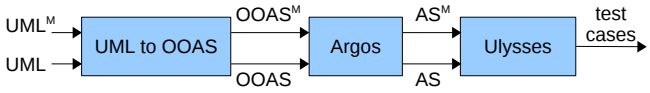


Figure 4. Test Case Generation Tool Chain.

As we want to create test cases that cover particular fault models, we need to deliberately introduce 'bugs' in the specification model. In order to do that, we rely on different mutation operators. As an example, one mutation operator sets guards of transitions to false[2], while other ones remove entry actions, signal triggers, or change signal events. Applying these operators to our CAS specification model yields 76 mutants: 19 mutants with a transition guard set to false, 6 mutants with a missing entry action, 12 mutants with missing signal triggers, 3 with missing time triggers, and 36 with changed signal events.

One particular mutant, shown in Figure 3 lacks the transition from *OpenAndUnlocked* to *OpenAndLocked*. This means that the system simply ignores the *Lock* event when being in the *OpenAndUnlocked* state instead of proceeding to *OpenAndLocked*.

Each mutant covers only one particular mutation (one mutation operation in a particular place): Mutation testing is based on the assumptions that (A) competent engineers write almost correct code, i.e. faults are typically "one-liners" and that (B) there exists a coupling effect so that complex errors will be found by test cases that can detect very small errors.

Ulysses cannot directly process UML models, but works on its own intermediate modelling language called Action Systems [5]. Hence, we first transform the UML diagrams of the original and mutated models into the input format of Ulysses. Figure 4 shows this tool chain. The first translation from UML to Object-Oriented Action Systems (OOAS) is done by the UMMU tool of AIT. UMMU also generates the set of mutants defined by its implemented mutation operators. Then, our tool ARGOS flattens an OOAS into a normal Action System (AS). The details of these model transformations have been described in [14]. Ulysses takes an original Action System model and one mutant as input and generates an abstract test case that will kill the mutant. This means that the test case is able to distinguish the original from the mutated model. Different strategies for deriving the test cases are available. The different strategies to kill a mutant have been presented in [1].

Here we will only use one strategy (A5), since the point of this paper is to show that generated test cases can be exchanged for deeper analysis. This strategy tries to minimise the number of generated test cases. Before creating new test cases for a mutated model, Ulysses first checks whether any of the previously created test cases is able to

[2] This may lead to a transition transforming into a self loop as the model will 'swallow' the trigger event (cf. Figure 3).

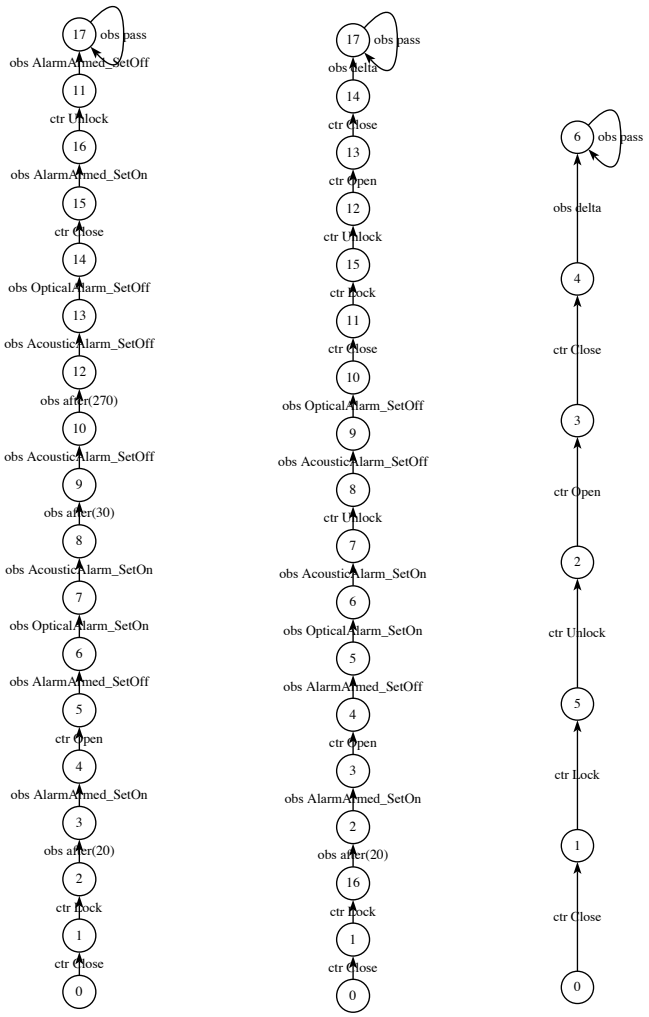


Figure 5. Three abstract test cases generated with Ulysses.

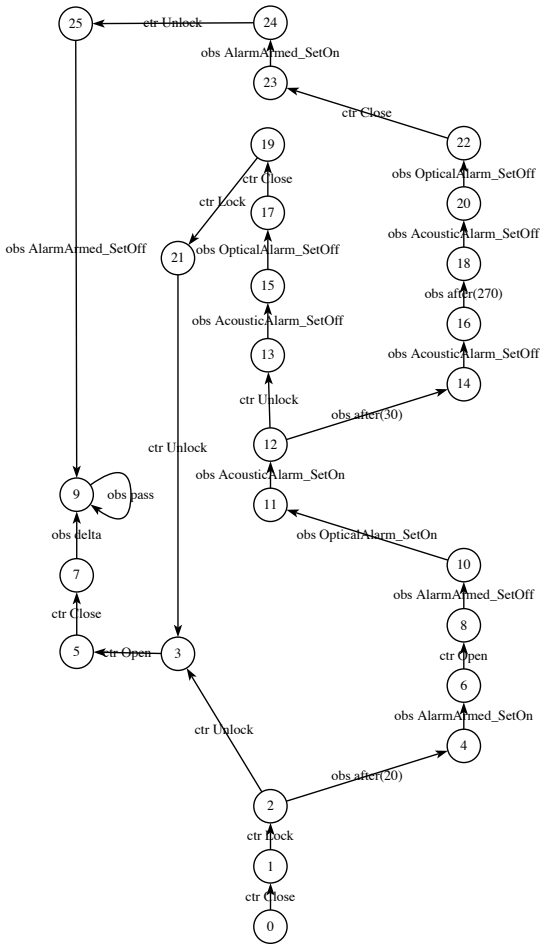


Figure 6. The three test cases of Figure 5 merged with the simplifiers of CADP.

kill the mutated specification.

With this technique, Ulysses generates 63 test cases in 23 minutes out of the 76 mutants. Figure 5 shows three of these abstract test cases. The labels with the prefix *ctr* denote controllable events (inputs), those with prefix *obs* denote observable events (expected outputs). The end of a test case is marked with a pass event. The absence of observations, called quiescence, is represented by a delta observation. In case of expected quiescence, an output must not occur, only inputs are possible. Next, we show how these test cases can be analysed in the CADP toolbox.

IV. Analysis in CADP

CADP is 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.

Our test case generator 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. In the following, we discuss how we merge the set of generated test cases into a single model.

A. Merging the 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]. Figure 6 shows the merged test cases of Figure 5. It shows that the merged test cases are combined into a trace-equivalent model.

When merging all the 63 test cases of the car alarm system, we obtain a trace-equivalent model with 50 states. 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. For example, common prefixes are merged into a single branch and can be easily spotted. In Figure 6 we immediately see that all three test cases start with closing the door. The possibility of first locking the door and then closing it is not tested!

B. Verification of Test Cases

CADP also 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 [15]. It is an extension of the alternation-free fragment of the modal mu-calculus [13] with action predicates and regular expressions over action sequences.

The idea is to formally verify certain properties of the merged test-case model. This serves two main purposes. First, we can check that our test cases are consistent with our requirements expressed as temporal properties. This allows the direct quality control of the test cases without relying on the models or test case generators. Our discussion with safety engineers showed that this is an important requirement for them. Second, since the test cases are generated from the model, we can detect faults in the models by detecting faults in the test cases.

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

```
[true* . "ctr Close" . (not "ctr Open")* . "ctr Lock"   (P1)
 .
 ((not "ctr Unlock") and (not "ctr Open"))*
 .
 "obs after(20)"
 .
 (not ("obs AlarmArmed_SetOn" or "obs pass"))
]
false
```

It partly formalises Requirement 1 and says that it must not happen (expressed by the `false` at the end) that the alarm is not switched to armed after 20 seconds when the doors are closed and locked. 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.

Similarly, Requirement 2 can be checked with a negative Property P2:

```
[true* . "obs AlarmArmed_SetOn" . (not "ctr Unlock")*   (P2)
 .
 "ctr Open" . "obs AlarmArmed_SetOff"
 .
 (not ("obs OpticalAlarm_SetOn" or "obs AcousticAlarm_SetOn"
       or "obs pass"))
]
false
```

It states that when the alarm is armed and an opening of the door happens, then first the armed signal is switched off and next one of the alarms is triggered.

So far, this guaranteed, that the test cases showed right behaviour according to the requirements. It was a soundness check. We can also check for test case completeness in the sense that we verify that certain traces are included in our test cases. For example the next Property P3 checks if a trace with first locking and then closing the doors leading to an armed state is included:

```
<true*><"ctr Lock"> <(not "ctr Unlock")*> <"ctr Close">   (P3)
<((not "ctr Unlock") and (not "ctr Open"))*>
<"obs after(20)"> <"obs AlarmArmed_SetOn"> true
```

Here the diamond operator `<.>` is used to express the existence of traces. It turns out that this property is not satisfied by our generated test cases. Hence, it seems that Ulysses never generated such a test case. An inspection of the (merged) test cases confirmed this. This shows that our integration with the CADP toolbox allows us to check if certain scenarios are included in a test suite. In the logic used, this scenarios can be expressed as regular expressions.

We can also highlight a possibly unwanted behaviour of our model with the following property that fails the model checking:

```
([true* . "obs AcousticAlarm_SetOff"                    (P4)
 .
 (not "obs AcousticAlarm_SetOn")*
 .
 "obs AcousticAlarm_SetOff"
]
 false)
```

Property P4 states that the acoustic alarm should not be switched off twice. It turns out that in the model the acoustic alarm is always switched off when the Alarm state is exited, although it might have been switched off already due to the 30 seconds timeout.

The purpose of this section was to motivate the verification of test cases generated from a model in a different tool. We can check requirement properties, ask if certain scenarios are included, or highlight additional properties that may be problematic.

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

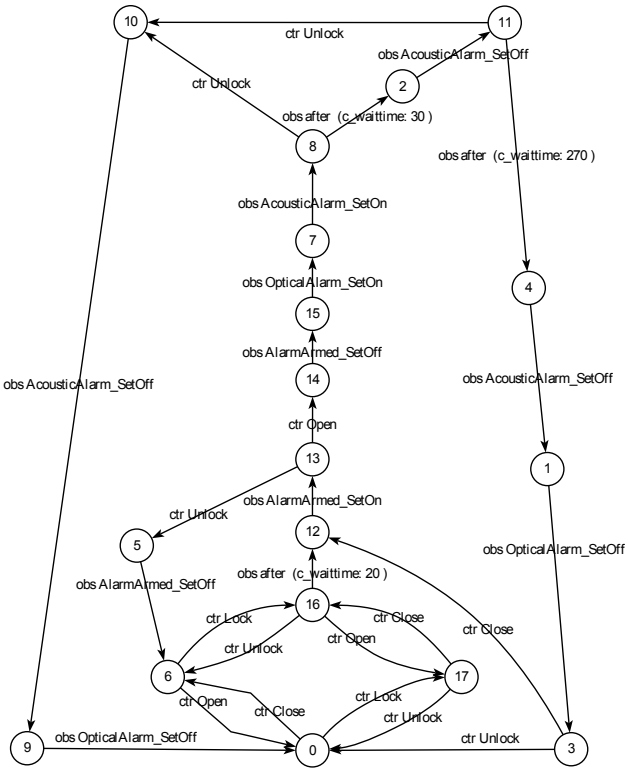


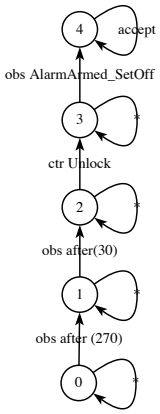
Figure 7. Labelled transition system of the car alarm system.

V. Generating Tests in CADP

Next, we are going to show that we can also go the opposite direction of integrating the two tools CADP and Ulysses. We will first generate test cases with CADP and then analyse how many UML mutants these test cases can kill.

Suppose we have a labelled transition system model of the car alarm system as shown in Figure 7. The CADP toolbox includes the test case generator TGV [11]. With this test case generator the tester can specify which test cases shall be selected from the model. Hence, the tester can steer the test selection process with so called test purposes. In TGV a test purpose is a deterministic input-output labelled transition system that is equipped with two sets of sink states, namely *Accept* and *Refuse*. The former defines the pass verdicts and at least one accept state must be present while the latter is used to limit the exploration of the graph during test case generation. A test purpose has to use the same alphabet as the specification model. TGV uses transitions labelled with a star ("*") in test purposes as a shorthand for "otherwise". Test cases are then generated "on-the-fly" from the synchronous product of the specification model and the test purpose.

Figure 8 shows a complex test purpose as a graph and in the input format of TGV. The test purpose asks for a test case that first exits the Alarm state via a time out and later




```
des(0,9,5)
(0,*,0)
(0,"obs after(270)",1)
(1,*,1)
(1,"obs after(30)",2)
(2,*,2)
(2,"ctr Unlock",3)
(3,*,3)
(3,"obs AlarmArmed_SetOff",4)
(4,accept,4)
```

Figure 8. A complex test purpose demanding a test case that exits the Alarm state twice.

exits the Alarm state via an Unlock event after the acoustic alarm has been switched off. Hence, a wanted test case enters and leaves the Alarm state at least twice. The star label at State 2 allows any sequence to happen before the final exit via Unlock. Hence, also several loops through the Alarm state are possible. However, in practise TGV will select the shortest possible test case satisfying the test purpose.

For our experiment, we designed 9 different test purposes by hand and let TGV create test cases. 3 out of the 9 test cases check for observable timeouts (time-triggered transitions: 20, 30, 300 sec. delay). Four test cases check the entry and exit actions of the states *Armed* and *Alarm*. One test case checks for the deactivation of the acoustic alarm after the timeout. The result of the complex test purpose of Figure 8 is a test case with a depth of 30 transitions going once through the state *SilentAndOpen* to *Armed* before going to *Alarm* again and leaving after the acoustic alarm deactivation by an unlock event. Hence, each observable event is covered by at least one test case. During the creation of the test purposes, we relied on a printout of the UML state machine.

Next we analyse these test cases in Ulysses.

VI. Mutation Analysis in Ulysses

The test cases generated by CADP-TGV can be imported into Ulysses. Ulysses can then perform a mutation analysis on our given set of 76 mutants. A mutation analysis shows which of the mutants were killed by a set of given test cases. Of course, we first check if the original UML model passes our imported test cases. This ensures that our LTS model for CADP is consistent with the UML model for Ulysses.

The mutation analysis of our 9 test cases from TGV shows the following results: Each analysis of a mutant takes about 5 seconds. Only 42 out of the 76 mutants can be killed by our 9 test cases designed via test purposes. This is a killing rate of 55%. A more detailed analysis presented in Table I shows the fault detecting power of the different test

Table I
MUTATION ANALYSIS OF THE NINE TEST CASES FROM TGV.

	acoustic alarm off	after20	after270	after30	alarm off	alarm on	armed off	armed on	complex	**Total**
Killed Mutants [#]	32	14	32	31	33	27	21	16	42	42
Killing Rate [%]	42	18	42	41	43	36	28	21	55	55

purposes. The first eight test purposes are named after the observation they are testing for. The ninth is the complex test case of Figure 8. It is the most efficient test purpose with respect to the killing rate. Its long test case kills 42 mutants which equals the total number of killed mutants. Hence, this test purpose subsumes all others. However, for debugging purposes it is still a good idea to keep the different test purposes with shorter test cases. Nevertheless, this mutation analysis shows that a greater variety in the test purposes is needed in order to kill more than 42 mutants. Such a mutation analysis could help to design better test purposes.

This analysis ends our tour from Ulysses to CADP and back.

VII. CONCLUSION

Summary. The purpose of this paper is to present and explore our idea of using test cases to integrate different model-based testing and analysis tools. We have generated test cases with our tool Ulysses that follows a model-based mutation testing strategy. These test cases were imported, merged and verified in the CADP toolbox. We checked required safety properties and verified that certain scenarios are missing from the test suite. Next, we generated test cases in CADP with the help of test purposes. We imported them into Ulysses and performed a mutation analysis. This showed that most of the test purposes were redundant. The main contributions of this work are (1) the new idea of using tests for tool integration in the given context and (2) the experiments with the integration scenarios between the two tools.

Discussion. The case study shows that this method may be a practical alternative for tool interoperability, where model transformations from one tool to the other are not yet available. The whole technique depends on model-based test case generation. The generated test cases represent a partial view on the model that can be analysed. With the right testing strategy the generated test suite may be seen as a representative abstraction of the original model. The failed verification of Property P4 showed that this may be sufficient for bug finding. However, the technique is very sensitive to testing criteria. The better the model is covered, the better it is represented by the test cases. For example, branch coverage will result in a similar abstraction as predicate abstraction. In this work, we used mutation coverage and test purposes. Of course, this approach will not replace full model verification.

As mentioned, the safety engineers we talked to are mostly interested in the test cases. The tests are needed for the certification processes. For the safety engineers, the models are a means to quickly generate high-quality test cases. However, from a certification point-of-view the models and test-case generators add a certain level of uncertainty to the whole development process. This is especially true, if the tools lack support for formal verification of the models. Therefore, the possibility to verify the test cases by importing them as a model into a verification tool may be of great help in practise. Further case studies are needed to support this thesis.

Related work. To our best knowledge our idea for tool integration by means of test case generation is novel. The closest related work that comes to our mind is model learning [3], [6]. In model learning a finite transition system is learnt from a finite set of queries to a system under test. The idea is to construct a formal model when no model is available. This is more difficult than generating tests from a given model and import them for analysis in another tool. However, with model learning algorithms we could reconstruct a better (smaller) model in the other tool after the import. The open question is if model learning would scale in this context. There are other standard minimisation algorithms that need to be investigated first, e.g. Brzozowski's algorithm.

The automotive industry has recently developed a common format for test case exchange: OTX (Open Test sequence eXchange) is a domain-specific language at a high level of abstraction with the aim of the graphic description of test sequences for the off-board diagnostics. It is available as ISO standard 13209. In diagnosis, test cases are used for fault localisation. In model-based diagnosis this is done on the modelling level. This can be seen as a tool integration based on test cases, but here the tests with actual outputs are used to resolve conflicts between the model and the actual observations. Hence, here the test cases are not used as a partial representation of the model but of the system under test. Nevertheless, there seems to be a growing awareness of the importance of exchanging test cases.

The future will show how far this idea can be taken in practise.

ACKNOWLEDGEMENT

The UML model and its mutants have been produced by Rupert Schlick, AIT in the MOGENTES project. 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

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

[2] B. K. Aichernig and J. He, "Mutation testing in UTP," *Formal Aspects of Computing*, vol. 21, no. 1-2, pp. 33–64, 2009.

[3] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, November 1987.

[4] ARTEMIS Industry Association, "ARTEMIS strategic research agenda 2011," 2011. [Online]. Available: http://www.artemis-ia.eu/sra

[5] R.-J. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," in *PODC'83, 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM, 1983, pp. 131–142.

[6] T. Berg, B. Jonsson, M. Leucker, and M. Saksena, "Insights to Angluin's learning," *Electronic Notes in Theoretical Computer Science*, vol. 118, no. 0, pp. 3 – 18, 2005, proceedings of the International Workshop on Software Verification and Validation (SVV 2003).

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

[8] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[9] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, July 1977.

[10] C. Hein, T. Ritter, and M. Wagner, "Model-driven tool integration with ModelBus," in *First International Workshop on Future Trends of Model-Driven Development, FTMDD 2009. Proceedings: In the context of the 11th International Conference on Enterprise Information Systems (ICEIS), 6-10th May 2009, Milan, Italy.* INSTICC Press, 2009.

[11] C. Jard and T. Jéron, "TGV: theory, principles and algorithms," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 4, pp. 297–315, 2005.

[12] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649 –678, Sept.-Oct. 2011.

[13] D. Kozen, "Results on the propositional μ-calculus," *Automata, Languages and Programming*, pp. 348–359, 1983.

[14] W. Krenn, R. Schlick, and B. K. Aichernig, "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)*, 2009, pp. 186–207.

[15] R. Mateescu and M. Sighireanu, "Efficient on-the-fly model-checking for regular alternation-free mu-calculus," *Science of Computer Programming*, vol. 46, no. 3, pp. 255 – 281, 2003, special issue on Formal Methods for Industrial Critical Systems.