Stefan Tiran

On the Effects of UML Modeling Styles in Model-based Mutation Testing

Master's Thesis

Graz University of Technology

Institute for Software Technology

Supervisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig

Graz, May 2013

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________________ ____________________________________

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ______________________________ ____________________________________

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Acknowledgments

I would like to thank many people who gave me support during this thesis.

First of all, I would like to express my gratitude to my supervisor Bernhard Aichernig. Already at the beginning of my master's program he gave me the chance to participate in the EU-FP7 project MOGENTES. I would also like to thank him for being so generous with his time and for setting up a weekly recurring meeting, in which I could ask him and he would answer all my questions.

I would like to thank the Austrian Institute of Technology (AIT) for employing me within the MBAT project for writing this thesis. Special thanks to Rupert Schlick for giving me technical advise on how to setup and use the *UMMU* part of the toolchain and also for helping me with the administrative stuff at AIT.

I want to thank Harald Brandl and Willibald Krenn, who introduced me into the MOGENTES project. I'm grateful to my current colleagues Birgit Hofer, Elisabeth Jöbstl, and Florian Lorber for the fruitful discussions and for being good friends.

I want to say thanks to my parents and grand parents for supporting me through all these years.

This thesis was funded from the ARTEMIS Joint Undertaking under grant agreement N$^{\text{o}}$ 269335 and from the Austrian Research Promotion Agency (FFG) under grant agreement N$^{\text{o}}$ 829817 for the implementation of the project MBAT, Combined Model-based Analysis and Testing of Embedded Systems.

Abstract

This thesis deals with the application of model-based mutation testing in software-development processes. In recent research projects the Austrian Institute of Technology and the Institute for Software Technology have developed a prototype toolchain, which can automatically derive test cases out of UML diagrams. In order to support modern software-development methods like test-driven development and enabling regression testing, the idea is used, to decompose test models into their functional components and gain partial test models. In later development phases, these partial models can be combined, which can be seen as refinement of the underlying partial models. In two case-studies this thesis shows, how partial models can be built. The first case-study deals with a car alarm system. A given test model is decomposed into two partial models and alternative modeling styles are presented. The second case study deals with the bucket control of an agricultural vehicle. Here, a given test model is optimized and a second partial model is introduced in order to cope with the complexity of the test case generation, so that it becomes computational feasible. Additionally, a comparison among different test case extraction strategies is conducted.

Kurzfassung

Diese Arbeit befasst sich mit der Anwendbarkeit von modellbasiertem Mutationstesten in der Softwareentwicklung. In vergangenen Forschungsprojekten wurde in Zusammenarbeit zwischen dem Austrian Institute of Technology und dem Institut für Softwaretechnologie an der TU Graz ein Prototyp entwickelt, der es ermöglicht, aus UML-Diagrammen automatisch Testfälle abzuleiten. Um den Bedürfnissen moderner Softwareentwicklungsmethoden wie der testgetriebenen Entwicklung gerecht zu werden und Regressionstesten zu ermöglichen wird die Idee aufgegriffen, die Testmodelle entsprechend ihrer funktionalen Einheiten aufzuteilen und dadurch partielle Modelle zu erhalten. In späteren Entwicklungsphasen können mehrere partielle Modelle zusammengefasst werden, was einer Verfeinerung (Refinement) der Ursprungsmodelle entspricht. In zwei Fallstudien wird gezeigt, wie ein Testmodell in mehrere partielle Modelle aufgeteilt werden kann. Die erste Fallstudie befasst sich mit einer Autoalarmanlage. Ein vorhandenes Testmodell wird in zwei partielle Modelle aufgeteilt und alternative Modellierungsstile werden gezeigt. Die zweite Fallstudie befasst sich mit der Steuerung der Baggerschaufel eines landwirtschaftlichen Fahrzeuges. Hier wird ein vorhandenes Testmodell optimiert und um ein weiteres partielles Modell ergänzt, um die Komplexität der Testfallgenerierung soweit in den Griff zu bekommen, dass eine fruchtbringende Anwendung von modellbasiertem Mutationstesten überhaupt stattfinden kann. Außerdem wird ein Vergleich unterschiedlicher Testfallextrahierungsstrategien durchgeführt.

Contents

List of Figures

List of Tables

List of Acronyms

AGSL	Activity and Guard Specification Language
AIT	Austrian Institute of Technology
CSP	Communicating Sequential Processes
DNF	Disjunctive Normal Form
ECU	Electronic Control Unit
EMF	Eclipse Modeling Framework
FSM	Finite State Machine
IST	Institute for Software Technology
LTS	Labeled Transition System
MBAT	Combined Model-based Analysis and Testing of Embedded Systems
MOGENTES	Model-based Generation of Tests for Dependable Embedded Systems
OCL	Object Constraint Language
OEM	Original Equipment Manufacturer
OMG	Object Management Group
OOAS	Object-Oriented Action System
RAISE	Rigorous Approach to Industrial Software Engineering
TFT	Thin-Film Transistor
UML	Unified Modeling Language
VDM	Vienna Development Method
VIATRA	Visual Automated Transformations for Formal Verification and Validation of UML Models

1 Introduction

1.1 Motivation

Testing is a very crucial part of software development, since it is the most common way of ensuring the quality and providing confidence. Modern approaches like test-driven development [Bec02] go so far to put testing in the center of the development, with test cases being created even before beginning to implement the system itself. While it can be considered as state-of-the-art to automate the process of executing test cases, creating them most often is still done manually. This is not only a costly procedure, but also the quality of the gained test cases varies. In order to improve the situation, research has been done on how to automate the generation of test cases. As a result, some prototype tools exist, which claim to perform such a fully automated test case generation. In recent research projects, a working prototype toolchain has been created in cooperation between the Austrian Institute of Technology (AIT) and the Institute for Software Technology (IST). It uses UML models as input and creates test cases using a so-called model-based mutation testing approach.

This thesis deals with the topic of modeling the requirements in a way that the models can be successfully processed by the toolchain.

1.2 Model-based Mutation Testing

1.2.1 Model-based Testing

Model-based testing is a general term for black-box testing techniques which use models as specification. In black-box testing as opposed to white-box testing the source code of the implementation is not available, so test cases have to be derived from requirements and specification documents. A textbook [UL07] and a journal paper [UPL12] written by Utting et al. provide a good overview of available technologies and approaches.

The approach that underlies the tools used in this thesis, can be classified as “Generation of test cases with oracles from a behavior model” according to Utting and Legeard [UL07]. A more detailed classification of the used techniques can be found in Section 2.1. The key point of these techniques is, that once the behavior model is given, the generation process can be automated.

According to Utting and Legeard the process for model-based testing consists of five steps:

1. Creating test models from requirements and specification documents
2. Choosing test selection criteria
3. Transforming the test selection criteria into test case specifications
4. Generating the test suite
5. Executing the test suite on the system-under-test

So instead of deriving test cases directly from the requirements, first an abstract model has to be created. Then the so-called test selection criteria have to be chosen. In a fault-based approach so-called "pre-specified faults" serve as criterion [UL07]. Test cases are created, which can determine, whether the anticipated faults have been implemented on the system-under-test. Since they are created using an abstract model, the generated test cases are called abstract test cases. In order to run them against the actual system-under-test, a test adapter and a test driver have to be written. This test adapter and test driver have to translate the abstract events of the test case into messages that are compatible to the interface of the implementation.

1.2.2 Mutation Testing

Originally, mutation testing has been introduced as method to measure the quality of test data which is used to test a program. It has been presented by DeMillo et al. [DLS78]. A comprehensive overview is given by Jia and Harman [JH11]. The idea is to create a set of so-called mutants, which are copies of the program but altered so that they contain simple errors. Then, the test data is applied to the original program and to each mutant. If the results of a specific mutant differ from the results of the original program, it is called a "dead" mutant, otherwise it is called a "live" mutant. Accordingly the given test data is able to "kill" a mutant or it is not.

There are two observations which support the power of this approach: the first is often referred to as *competent programmer hypothesis*. In DeMillo et al. write:

> "Programmers have one great advantage that is almost never exploited: they create programs that are *close* to being correct!"

The second observation is called *coupling effect*:

> "*The coupling effect:* Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors."

Both observations have been investigated in more detail by Offutt [Off92]. In his work also a definition for the difference between simple and complex errors resp. faults is given. According to Offutt, a fault is simple if it can be repaired by a single change, otherwise it is complex. He also distinguishes between "simple mutants", which are obtained by applying a single mutation and "complex mutants", also called "higher-order mutants" which are obtained by applying multiple mutations. Thus simple mutants are simple faults, while complex mutants are complex faults.

As a consequence it is assumed, that simple mutants are sufficient. Thus, the mutation operators, which define which syntactical element of the source code is replaced in the mutant create a new copy for each mutation.

1.2.3 Model-based Mutation Testing

Model-based mutation testing is the application of mutation testing as test selection criterion for model-based testing. Instead of mutating a program, the mutation is performed on the test model. As in classical mutation testing *mutation* means the creation of a copy of the model which differs in just one position. This is done by implementing "mutation operators". Each mutation operator scans the original model for a specific syntactical element and replaces it, so that it is still syntactical correct, but the behavior changes. The application of these mutation operators to the original model leads to a set of mutants. Together they form a fault-model, which can be seen as test case specification. As in classical mutation testing, the set of mutants can either be used to measure the "fault-finding power" of a given test suite, or they can be used to "guide the design" of new test cases. [UL07]

When used in the test-case generation process the goal is to create and select test cases, which are able to reveal the difference between the original test model and the derived mutants.

So in summary the aim of model-based mutation testing is to create an abstract test model, for which *mutation operators* anticipate a comprehensive set of simple faults. In a test case generation process test cases are generated, which can decide whether the anticipated faults have been implemented on a given system-under-test or not. Because of the *coupling effect* it can be argued that the absence of the simple faults in an implementation also implies the absence of more complex faults.

1.3 UML

The Unified Modeling Language (UML) [BRJ05] is a visual modeling language managed by the Object Management Group (OMG). Its main purpose is to provide a common language for visualizing and communicating software blueprints for object-oriented systems, where it has become a de facto standard. It provides several different diagram types, which allow to model different aspects of a system. There are many software tools supporting UML, most of them for the purpose of creating design models of software under development. However, there also exist various model-based testing tools, which take UML models as input. The advantage of using UML is that it is widely known and the diagrams are often perceived as easy to read. The disadvantage of using UML as input models for model-based testing is, that there are some semantic variation points, which are underspecified by the specification. Because of that, UML often is criticized as being only semi-formal. In order to use UML models for formal methods, a concrete subset of UML features has to be defined and a precise semantics has to be given.

1.4 Refinement, Partial Models, and Compositionality

A widely known definition of *refinement* is given by the ISO reference model for open distributed components [ISO97]. There the term *refinement* refers to two concepts:

- The process of obtaining a more detailed specification.
- The relation between an original specification and a more detailed specification.

The detailed specification is said to refine an original specification only if they are behaviorally compatible. Two objects are behaviorally compatible if the environment can not distinguish them.

Note, that these definitions are rather weak and depend on user-defined criteria. For this thesis refinement is based on the conformance relation *ioco*, which is described in Section 2.2.

The idea of refinement in general, and "step-wise refinement" in particular is linked to a formal software-development method. From a requirements document an initial abstract model of the system is built. Later, in so-called "refinement steps" this model becomes more concrete until finally, an implementation can be derived. One early adopter of this method is VDM [Jon90] or more recently the B-method [Abr96] and its successor Event-B [Abr10]. The latter is supported by a software platform called "Rodin", which allows the formal check of each refinement step using a proof system, that guarantees the implementation satisfies its initial specification [Abr+10].

The idea of partial models in general is to avoid being too concrete when creating abstract models in early refinement steps. One early approach to implement this idea is the introduction of "Modal Transition Systems" as proposed by Larsen and Thomsen [LT88]. In this logic, a distinction between "necessary" and "admissible" actions is made explicitly. When using *ioco* as conformance relation, all models are inherently partial, since the *ioco* relation only cares about whether outputs from the SUT are allowed. Inputs which are not defined by the specification are implicitly considered as underspecified, allowing the SUT to react arbitrarily. Additionally, underspecification of output behavior can be achieved by explicitly using the concept of a "non-deterministic choice". A detailed discussion with examples can be found in Section 4.1.

Partial models also allow to decompose a system into functional components. This leads to the idea of so-called "compositional testing" as proposed by the group of Tretmans [BRT04]. The idea is to imply the correctness of an integrated system from the correctness of the parts. In the context of this thesis, the term "partial model" will be used only to denote test models, which only contain some functional components.

1.5 Project Background

This thesis is based on the outcomes of the EU-FP7 project MOGENTES[1]. MOGENTES is an acronym for "Model-based Generation of Tests for Dependable Embedded Systems". The project took place from January 2008 to March 2011 and had been coordinated by AIT. It consisted of ten organizations including research centers and university institutes conducting the research and industrial companies providing demonstrators of the automotive and railway domain.

The prototype toolchain MoMuT as well as both demonstrators used for the case studies are artifacts produced within the project. Also the author of thesis has been employed within this project for nine months.

For writing this thesis the author has been employed by AIT within the Artemis project MBAT[2], which is an acronym for "Combined Model-based Analysis and Testing of Embedded Systems". The aim of this project is to integrate existing tools and technologies for performing model-based validation and verification into a "Reference Technology Platform". The project runs for 36 months and consists of 38 organizations. The demonstrators within this project focus on embedded systems in airplanes, cars and trains.

1.6 Running Example

To illustrate the algorithms and concepts of the theoretical part of this thesis and of the description of the toolchain, examples are used which are based on one of the case studies.

This case study investigates a car alarm system. The specifications come from Ford and have been provided as demonstrator within the FP-7 project MOGENTES.

The core requirements of this system are:

R1 - **Arming** "The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment, and all doors are closed.

R2 - **Alarm** The alarm sounds for 30 seconds if an unauthorized person opens the door, the luggage compartment, or the bonnet. The hazard flasher lights will flash for five minutes.

R3 - **Deactivation** The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside. " [KSA09]

Since these requirements leave some decisions open, several test models haven been written in UML by AIT. They differ about the interpretation of these requirements.

One particular model has been used as example in earlier published work of the project [KSA09]. There the translation from UML to object-oriented action systems is described.

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

[2] `http://www.mbat-artemis.eu/`

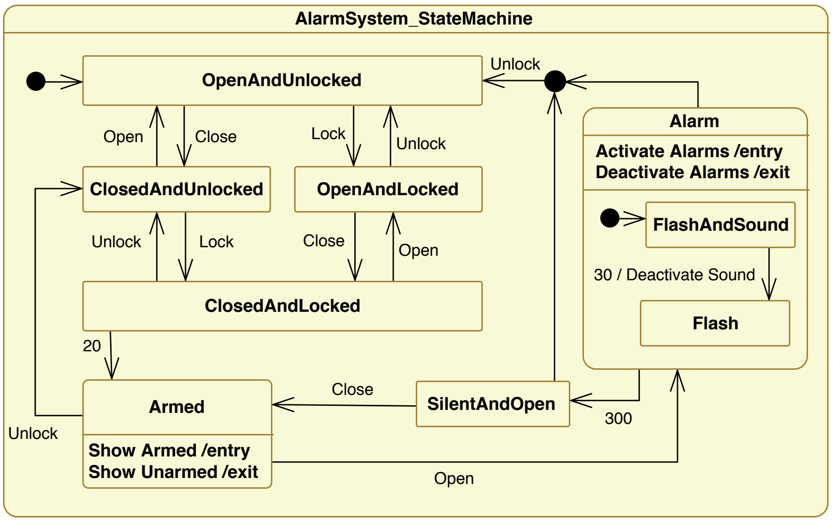


Figure 1.1: State machine of the car alarm system used in [KSA09]

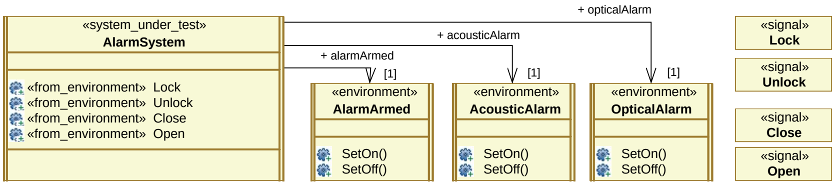


Figure 1.2: Class diagram of the car alarm system used in [KSA09]

This model is shown in Figure 1.1, depicting the state machine and Figure 1.2 depicting the class diagram, which defines the system boundaries and therefor the test interface. There are four inputs to the system (“Lock”, “Unlock”, “Close”, and “Open”), which are modeled as signals. The outputs to the actuators are divided into three classes “AlarmArmed”, “AccousiticAlarm”, and “OpticalAlarm”. They each provide two operations “SetOn” and “SetOff”, which correspond to the outputs responsible for the arming, the sound alarm, and the hazard flasher lights resp. A description for the used stereotypes can be found in Section 3.1.2.

The transitions within the state machine are labeled with the triggers firing them and the effects they cause. In this particular model two kinds of triggers are used: signal receptions and time triggers. The signal receptions are labeled according to the signal, which is received as input and the time triggers are labeled with the time, after which the transition is taken. Outputs produced by the system are modeled in so-called “opaque behaviors” which are either used in effects or in entry- and exit actions. The according labels are “Armed”, “Unarmed”, “Activate Alarms”, “Deactivate Alarms”, and “Deactivate Sound”.

Being very well sized this demonstrator has been chosen as default example of our workgroup. It is small enough to be analyzed even by unoptimized test case generation algorithms in reasonable time, but yet not trivial, since it contains interesting features like time constraints.

1.7 Aims and Goals

This thesis investigates the effects of different modeling styles in test models for a particular prototype toolchain performing model-based mutation testing. Using class diagrams and state machines of UML models as input language, there are several possible ways how to build a test model.

One particular concept is the use of partial models and compositional testing in model-based mutation testing. This approach is based on ideas, which are also presented in our technical report [ALT12b], where a novel formal test-driven development process is proposed. If at the beginning of the development process first partial test models are created, which are later refined, then test-driven development can profit most from model-based testing.

In two case studies the use of partial models is demonstrated.

In the first case study, the already mentioned car alarm system is investigated. For this demonstrator it is shown how a test model can be decomposed into partial models. This is done by creating test models using different modeling styles. Doing so gives a good overview over the capabilities and limitations of the used toolchain.

In the second case study, a wheel loader is investigated. Within the MOGENTES project some test models have been created, which could be used to generate meaningful test cases using the *MoMuT* toolchain. However, the results had not been satisfying. Processing all mutants of a test model took more than a month. To make things worse, a test suite generated using a random walk in a fraction of this time had been able to outperform the mutation-based test suite. One aim of this thesis is to improve the test models applying the ideas of refinement and partial models in order to generate test suites using the *MoMuT* toolchain in reasonable time outperforming random test case generation. The partial models shall be used to repeat the experiments regarding the comparison among different killing strategies [Aic+11a] in order to gain empirical results for further publications.

1.8 Structure

This thesis is structured as follows:

Chapter 2 introduces the theoretical background of this thesis. Here all the used concepts and notations are introduced. Section 2.1 relates the approaches used by the tools to a taxonomy of model-based testing. Section 2.2 explains the conformance relation *ioco* which is the part of the testing theory, which is crucial for the design of test models. Section 2.3 describes the notion of action systems which serve as intermediate language.

Chapter 3 discusses the used toolchain and its development throughout the MOGENTES project. Also implementational details which influence the creation of the test models are discussed.

Chapter 4 describes the implications a model-based mutation testing approach has on the development process. The main focus lies on the ideas of refinement, how partial models and compositionality can help to create feasible models, which are able to provide meaningful test suites at the right time. Section 4.1 shows, which features of the conformance relation *ioco* enable the use of partial models and where the limits are. Section 4.2 relates the proposed way of refinement based on partial models to other forms of refinement. Section 4.3 explains how partial models in a model-based testing approach can help to generate test cases in early development phases. So combining these techniques can lead to a new test-based development process.

Chapter 5 presents the car alarm case study. Section 5.1 describes how the test model is decomposed into partial models and why it makes sense to use non-determinism. Section 5.2 presents three alternative UML modeling styles applied on the car alarm system. Section 5.3 shows the results from performing the test case generation of the presented test models and the mutation analysis used to measure the quality of the generated test suites.

Chapter 6 presents the wheel loader case study. Section 6.1 describes the requirements of the demonstrator in detail. Section 6.2 shows both the existing test model and its improvements as well as the new additional partial model. Section 6.3 shows the results from performing the test case generation and the mutation analysis.

Chapter 7 gives concluding remarks. Section 7.1 summarizes the content of this thesis. Section 7.2 shows related work on partial models. Section 7.3 discusses the results from the case studies. Section 7.4 gives an outlook on future work.

2 Prerequisites

2.1 Classification of Model-based Testing

For classifying the different available model-based test case generation techniques, the survey by Utting et al. [UPL12] provides a taxonomy with six dimensions. In the following this taxonomy is used to compare the toolchain used in this thesis to other possible technologies.

The first dimension is the so-called “model scope”. It defines, whether inputs and outputs or only inputs are modeled. This has a high impact on the so-called “test oracle”, which decides, whether or not a system passes a test case. If outputs are omitted, the oracle is considered as “weak oracle” since it only focuses on whether the system crashes or raises an exception. The approach used in this thesis requires to also model the desired outputs, which leads to a much stronger oracle.

The second dimension are the so-called “model characteristics”. This dimension includes decisions made on time handling, determinism vs. non-determinism, and an event-based view vs. the use of continuous systems. The *MoMuT* toolchain supports basic time-outs and non-determinism and uses an event-based view. Note, that the term “non-determinism” is ambiguous. In this thesis, the term is used in the sense of underspecification. This is discussed in detail in Section 2.2. The complete toolchain only supports an event-based view. However, some parts, like the test-case generator *Ulysses*, described in Section 3.4 are already prepared for hybrid systems.

The third dimension is the so-called “model paradigm”. For this thesis, a combination of different paradigms is used: On the one hand, there is a strong focus on so-called “Transition-Based Notations”, since UML state machines are used, on the other hand there is also heavy use of “State-Based Notations”. This is very important for this thesis, since it allows for different modeling styles. One important contribution of this thesis is the investigation of the impact of the modeling style on the test-case generation.

The fourth dimension are the so-called “test selection criteria”. As the title of this thesis already suggests, mutation testing, which classifies as fault-based selection criterion, is used. Mutation operators are used to create a set of faulty models, which are later analyzed in order to create test cases. Depending on which mutation operators are used, this test selection criterion can also cover other criteria. For instance, in the toolchain used in this thesis, there is one mutation operator, which corresponds to ‘transition coverage”, which is considered a “structural model coverage criterion”.

The fifth dimension is the so-called “test generation technology”. The toolchain used in this thesis can be classified to do “bounded model checking”. However, the properties, which are checked are not formulas in temporal logic, but instead the conformance between a mutated model and the unmutated model is checked.

The sixth dimension is the decision between online and offline testing. In online testing, the test tool communicates directly to the system-under-test, whereas in offline testing a test suite is created, which can later be executed using a test adapter and a test driver. The toolchain used in this thesis performs offline testing.

In summary the toolchain used in this thesis is based on an approach operating on *non-deterministic input-output* model specifications, that are *transition-based* in the front end and have a *pre-post* structure in the internal representation to generate *offline* tests that cover *fault-based* criteria, which can subsume *structural* criteria by applying *model checking*.

2.2 The Conformance Relation ioco

2.2.1 Definition

The toolchain used for this thesis is designed for the conformance relation *ioco*. It has been presented by Tretmans [Tre96]. Using *ioco* allows for partial models and non-determinism in the sense of underspecification. To decide the conformance $i \, \mathit{ioco} \, s$ between the implementation i and the specification s, both implementation and specification have to be presented as Labeled Transition System (LTS). Note, that the terms *implementation* and *specification* refer to the role in conformance checking and not to the level of abstraction. In fact, for generating test cases the tool can use mutated models as implementation or even existing test cases as specifications, since they can all be expressed as LTS.

The definition of a labeled transition system that is used in earlier work of our workgroup [Aic+11a], where the approach of the toolchain is described, is:

> “Thereby, a labeled transition system is defined as tuple $\langle S, L, T, s_0 \rangle$ where
>
> - S is a countable set of states
> - $L = L_U \cup L_I$ is a countable set of labels divided into input labels L_I and output labels L_U such that $L_I \cap L_U = \emptyset$
> - $T \subset S \times (L \cup \{\tau\}) \times S$ is the transition relation, and
> - $s_0 \in S$ is the initial state.”

So the labels denoting events of the system are distinguished between input and output events. Additionally there are internal events, which are expressed by the special label τ, which is neither an input nor an output label.

The system serving as implementation is considered to be weakly input-enabled, which means that all input events are possible in any state and the implementation is not allowed to prevent them. However, it may be necessary to take internal transitions before. Technically, input enabledness is realized by adding self-loops with otherwise undefined inputs. The class of LTS which are input-enabled is called IOTS.

For each state of the LTS L, where no output is produced, a self-loop with a special output label denoting *quiescence* is added to the LTS. After determinizing this automaton this results in a so-called suspension automaton $\Delta(L)$. The set $Straces(s)$ contains all possible traces through the suspension automaton, which are called suspension traces.

$$Straces(L) =_{df} traces(\Delta(L))$$

A suspension trace is a sequence of input- and output-events including the special output δ for quiescence. For each IOTS and each trace the function *after* returns the set of states that are reached, when following the labels of the trace. The function *out* takes a set of states and returns the set of output labels that occur in transitions going away from any of the given states.

Finally $i\, ioco\, s$ can be defined as:

$$i \text{ } ioco \text{ } s =_{df} \forall \sigma \in Straces(s) \bullet out(i \text{ } after \text{ } \sigma) \subseteq out(s \text{ } after \text{ } \sigma)$$

The rationale behind this definition is that conformance only depends on traces, defined by the specification. This has the great advantage that partial models can be used to specify a system and a full model or an implementation can conform to it. All input-events that are not part of the specification are left to the freedom of the implementation. However, if a trace is defined by the specification, then in order to conform, any output-event (either an output label or quiescence) occurring on the implementation has to be allowed by the specification. Any allowed output-event after a trace σ is element of the set *out*(s *after* σ), so the set of performed output-events by the implementation *out*(i *after* σ) has to be a subset.

If the specification LTS models a deterministic system, each set of allowed outputs after any trace contains exactly one output event, so in order to conform to the specification the implementation has also to be deterministic and produce the same output as the specification. If on the other hand the specification LTS models a non-deterministic system, there are some traces after which the set of allowed outputs contains more than one element and it is up to the implementation which one to produce. This means, that non-determinism is always understood in the sense of underspecification and there is no fairness-assumption which would require an implementation to be non-deterministic, too.

Example To illustrate the meaning of *ioco* conformance let us consider the following example:

Given the LTS S_1, S_2 and I, which represent two different specifications and one implementation for a minimized version of the car alarm system. Let the alphabet consist of the set of input-labels $L_I = \{ \mathit{Close} , \mathit{Lock} \}$ and the set of output-labels $L_U = \{ \mathit{ArmedOn} \}$. Consider the set of states, the transition relation and the initial state as visualized in Figure 2.1

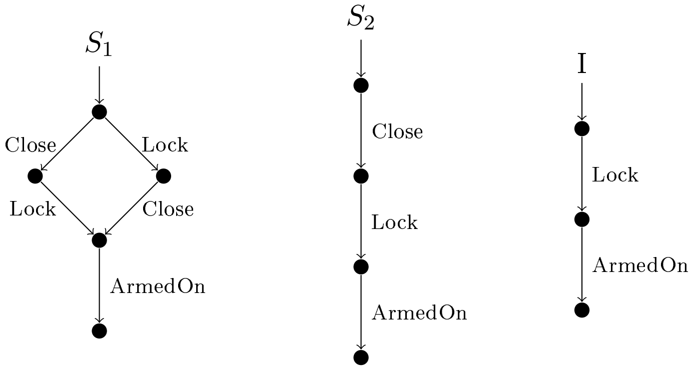


Figure 2.1: LTS of *ioco* example

In a first step, the implementation becomes input-enabled by adding self-loops with all elements of L_I at each state, where a transition with an input label is missing. This leads to the IOTS depicted in Figure 2.2.

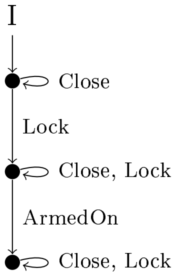


Figure 2.2: IOTS of implementation of *ioco* example

In a next step the suspension automata are derived, as depicted in Figure 2.3.

To decide, whether $I \, ioco \, S_1$ and $I \, ioco \, S_2$ hold, the traces of S_1 resp. S_2 have to be investigated.

Table 2.1 shows some traces, which can be used to instantiate the formula. In the last row, the trace $<$ Lock $>$ is considered. After this trace S_1 prescribes quiescence, which is violated by the implementation by producing the output event “ArmedOn”. This is a counterexample, so $I \, ioco \, S_1$ does not hold.

Table 2.2 shows the relevant traces for determining whether $I \, ioco \, S_2$ holds. Note, that traces with consecutive quiescence have been omitted, as otherwise the number of traces would be infinite. So technically Table 2.2 is not a proof that $I \, ioco \, S_2$ holds. However, it can be argued, that since both models represent a deterministic system, these traces

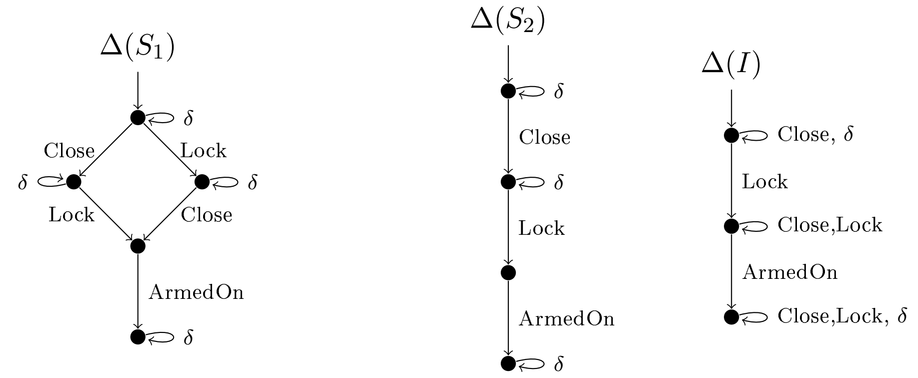


Figure 2.3: Suspension automata of *ioco* example

σ	out(I *after* σ)	out(S_1 *after* σ)	$\subseteq$
$<>$	$\{\delta\}$	$\{\delta\}$	✓
$<$ Close $>$	$\{\delta\}$	$\{\delta\}$	✓
$<$ Close , Lock $>$	$\{ArmedOn\}$	$\{ArmedOn\}$	✓
$<$ Lock $>$	$\{ArmedOn\}$	$\{\delta\}$	✗

Table 2.1: Traces to decide I *ioco* S_1

σ	out(I *after* σ)	out(S_1 *after* σ)	$\subseteq$
$<>$	$\{\delta\}$	$\{\delta\}$	✓
$<$ Close $>$	$\{\delta\}$	$\{\delta\}$	✓
$<$ δ, Close $>$	$\{\delta\}$	$\{\delta\}$	✓
$<$ Close , Lock $>$	$\{ArmedOn\}$	$\{ArmedOn\}$	✓
$<$ δ, Close , Lock $>$	$\{ArmedOn\}$	$\{ArmedOn\}$	✓
$<$ Close , δ, Lock $>$	$\{ArmedOn\}$	$\{ArmedOn\}$	✓
$<$ δ, Close, δ , Lock $>$	$\{ArmedOn\}$	$\{ArmedOn\}$	✓
$<$ Close , Lock , ArmedOn $>$	$\{\delta\}$	$\{\delta\}$	✓
$<$ δ, Close , Lock , ArmedOn $>$	$\{\delta\}$	$\{\delta\}$	✓
$<$ Close , δ, Lock , ArmedOn$>$	$\{\delta\}$	$\{\delta\}$	✓
$<$ δ, Close, δ , Lock , ArmedOn $>$	$\{\delta\}$	$\{\delta\}$	✓

Table 2.2: Traces to decide I *ioco* S_2

won't reveal non-conforming behavior either. Another way to determine the *ioco* conformance is shown in Section 2.2.2.

2.2.2 Product Graph

For automating the decision, whether for two LTS i and s the statement $i\,\mathit{ioco}\,s$ holds, Weiglhofer and Wotawa proposed the idea of a synchronous product $\times_{ioco}$. The result of $i \times_{ioco} s$ is a LTS which contains a special "fail" state if and only if $i\,\mathit{ioco}\,s$ does not hold. Additionally a synchronous product contains a so-called "pass" state for allowed behavior of the implementation that is underspecified by the specification.

In later work of the workgroup [BWA10] this was changed slightly so that now "fail" resp. "pass" states are regular states, with the property that they are sink states with a self-loop, which has a special label "fail" resp. "pass". This kind of synchronous product is also referred to as "product graph".

The *product graph* $i \times_{ioco} s$ can be constructed automatically by exploring both LTS simultaneously and applying the following rules: [BWA10]

- If a transition is possible in both i and s, it is added to the product graph.
- If a transition with an input label is possible in i but not in s, then a new node is created in the product graph. Then a transition with said input label leading to this node is created and a self-loop with the special label "pass" is added.
- If a transition with an output label is possible in s but not in i, then also a new node, with a "pass" self-loop is created and is connected to the graph with a transition using said output label.
- If a transition with an input label is possible in s but not in i, then it is assumed that i contains a self-loop with this input label and thus stays in the same state.
- If a transition with an output label is possible in i but not in s, then a new node with a "fail" self-loop is created and connected to the graph using said output label. In this case non-conforming behavior has been revealed.

Example As example consider the three LTS from Figure 2.3, which have already served as example in Section 2.2.1. Figure 2.4 shows the corresponding product graphs. Since $I \times_{ioco} S_1$ contains a *fail* transition and $I \times_{ioco} S_2$ does not, $I\,\mathit{ioco}\,S_1$ does not hold, while $I\,\mathit{ioco}\,S_2$ does.

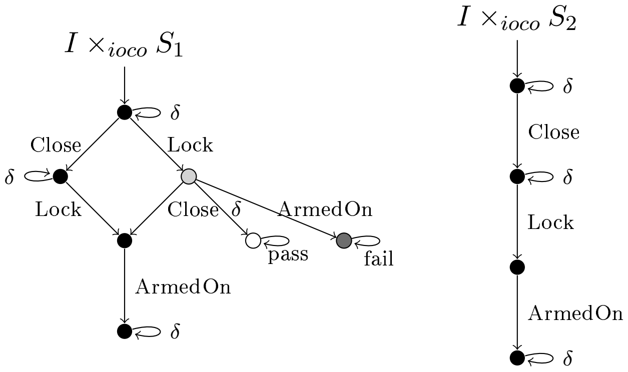


Figure 2.4: Product graphs of *ioco* example

2.3 Action Systems

2.3.1 Flat Action Systems

Instead of directly mapping the UML input models to LTS, in the *MoMuT* toolchain two intermediate languages are used. They are based on the action system formalism proposed by Back and Kurki-Suonio [BK83]. First the UML model is translated into an object-oriented action system, which is later translated into a flat version of an action system. This process has been described in previous publications of this workgroup [Aic+11b].

There the abstract syntax of Action Systems have been defined as:

$$\begin{array}{llllll} & AS & =_{df} & |[& \textbf{var} & V : T = I \\ & & & & \textbf{functions} & F_n^1 = F_b^1; \ldots; F_n^m = F_b^m \\ \text{``} & & & & \textbf{actions} & N_n^1 = N_b^1; \ldots; N_n^a = N_b^a \quad " \\ & & & & & \textbf{do}\ A_1 \square \ldots \square A_d\ \textbf{od} \\ & & &]| : F_I & & \end{array}$$

Action systems are state based. The state is represented by a vector of variables V, which are initialized with the vector I. The state update can be done either in the named actions or they can be swapped out into functions. The basic concept of a state variable update is a so-called guarded command, which has been introduced as guarded iteration statement by Dijkstra [Dij76]. The rationale behind this notion is that each action has a so-called guard, which is a condition over the variables, that decides, if the action is enabled. The operator $\square$ is called “non-deterministic choice” and states that in each iteration of the loop one action is chosen non-deterministically. Within this loop, $A_1 \ldots A_d$ are so-called anonymous actions. They can consist of basic actions and additionally of calls of the named actions $N^1 \ldots N^a$.

The following basic actions exist: [Aic+11b]

Guarded Command. A guarded command has the form “requires p: S end” where p is the condition serving as guard and S can be any other basic action. That means, that guarded commands can be nested.

Assignment. An assignment is used to actually perform a state update and has the form “$y := e$” where y is a variable and e is an expression.

Local Variables. Local variables can be introduced to store intermediate results that shall not introduce a new state. The general form is “var x_1: $T_1; \ldots; x_n$: T_n: S” where $x_1 \ldots x_n$ are the introduced variables of type $T_1 \ldots T_n$ and S is any other basic action.

Sequential Composition. A sequential composition has the form “$S_1; \ldots; S_n$” and is used to denote that the basic actions $S_1 \ldots S_n$ are performed in the given order. Note, that since the composed action is performed atomically, the sequential composition of some basic actions is only enabled if every single action is enabled. So if a basic action is not enabled it hinders all basic actions which are connected using the sequential composition, even if they occur before it.

Nondeterministic Composition. A non-deterministic composition has the generic form "$S_1 \quad \Box \; S_2$" and denotes the same as the *non-deterministic choice.* Either S_1 or S_2 are performed, if they are enabled.

Prioritized Composition. The prioritized composition has the generic form "$S_1 // S_2$" and denotes that if S_1 is enabled, it is performed, otherwise S_2 is performed. It is merely a short form for a non-deterministic composition and a guarded command consisting of the negation of the guard-condition of S_1 as guard for S_2.

Skip. Skip is the basic action used to denote an empty assignment, leaving the state unchanged.

Named actions have a name and a body. The name may be prefixed with "ctr" or "obs" to mark an action as input resp. output action, otherwise the action is treated as internal. This is important, since the name of the actions correspond to the labels in the LTS, and labels are distinguished between input and output labels as described in Section 2.2.1.

The body of a named action can consist of basic action as well as of calls of functions, but not other named actions. Functions can only consist of basic actions. That way, there are no direct or indirect recursive calls, and for evaluating the action system all calls can be inlined.

2.3.2 Object-Oriented Action Systems

Additionally to the flat action systems which serve as input language for the *ioco* checker *Ulysses* there are so-called object-oriented action systems which serve as intermediate modeling language. In this language an object-oriented extension is made to action systems. This extension is a limited implementation of an approach proposed by Bonsangue et al. [BKS98].

An object-oriented action system may consist of several action systems which are treated as classes. So it should not surprise that the definition of a class resembles the definition of an action system:

$$\text{``}\quad \begin{array}{lllll} C_b & =_{df} & |[& \textbf{var} & V : T = I \\ & & & \textbf{methods} & M_n^1 = M_b^1; \ldots; M_n^m = M_b^m \\ & & & \textbf{actions} & N_n^1 = N_b^1; \ldots; N_n^a = N_b^a \\ & & & \textbf{do } A \textbf{ od} & \\ & &]| : M_I & & \end{array} \quad \text{'' [Aic+11b]}$$

The current implementation of the *MoMuT* toolchain (the responsible part is called *Argos* and described in Section 3.3) only allows for a finite set of classes and a finite set of objects. So objects can only be created within the **var** block. Additionally one class is marked as *autocons*, which means that one object of this class is created and serves as root object.

To prioritize the actions between objects of different classes, a so-called “system assembling block” is introduced. There all classes used in the object-oriented action system are enumerated and connected using either the $\Box$ or // operator. When the // operator is used to connect two classes, actions of the latter class or only enabled if none of the former is.

Communication is done via a shared memory. Technically, this is realized by getter and setter methods, since methods can be called from actions of any class.

Variables can either be objects, Booleans, Integers or enumerations. For Integers the typing system is very strong including lower and upper bounds. This is due to the enumerative exploration approach done by *Ulysses*, which is very sensitive about the range of values that can be used as parameters. Additionally there is support for lists and tuples, which can be of any defined type, including other list types and tuples. The strong typing system prescribes to include the length of a list in its definition.

3 The Toolchain

All experiments described in this work are conducted using the prototype toolchain, which has been created within the MOGENTES project with the purpose of test case generation. This toolchain is currently named *MoMuT*. Throughout the development it has also been referred to as the UML/OOAS track, since it uses UML as input language for models which are then transformed into so-called Object-Oriented Action Systems.

Figure 3.1 gives an overview over the tools contained in this toolchain. The solid boxes represent the programs, those within the dashed box are considered as part of the toolchain. The lines with arrows represent files, which are created as output by one program and used as input by the next program. The label *OOAS* refers to the object-oriented action system encoding the original model and the label $OOAS^M$ refers to the object-oriented actions systems encoding the model mutants. The labels *AS* and AS^M refer to the resp. flat action systems.

The tool *Argos* has to be run once per mutant and once with the original model. Also the tool *Ulysses* has to be run for each mutant independently. Shell scripts are used to automate this process.

3.1 Input Language

Even though object-oriented action systems are a modeling language of their own, the designated input language of the toolchain is UML. In the MOGENTES project the Eclipse Modeling Framework (EMF) has been used as common representation format. As formalism a subset of UML 2.0 and OCL 2.0 is used. Supported elements of UML are class diagrams and state machine diagrams. [Pol09]

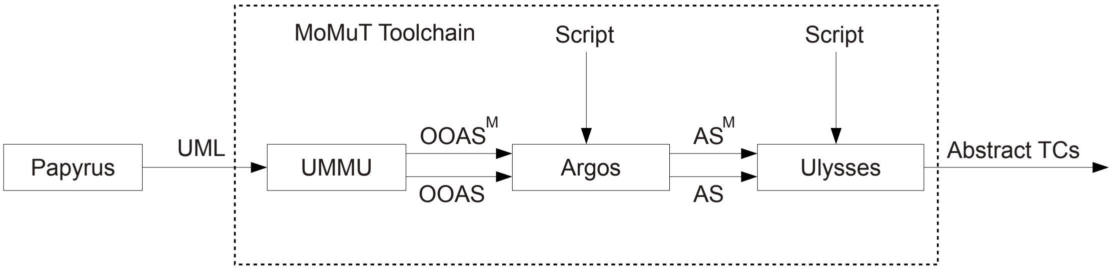


Figure 3.1: Overview over the MoMuT toolchain

As modeling tool Papyrus UML[1] is used. [Kro10]

3.1.1 Used UML elements

UML specifies many different diagrams and for each diagram many different elements, but not all of them are supported by the *MoMuT* toolchain. In fact, only two types of diagrams are used: *class diagrams* and *state machine diagrams*

One of the core functions of the class diagrams used for test models is to define the test interface. That means, class diagrams are used to define the system boundaries and to determine which events can occur in the generated test cases. Both environment and the system-under-test are represented as classes. The events, which are used in the test cases, can either be modeled as signal receptions or operations.

Signal receptions in a class representing (a part of) the system-under-test are considered as inputs, while operations and signal receptions in an environment class are considered as output. Both signals and operations can have parameters. Properties of classes can be used as variables, which can be accessed by OCL constraints and so-called opaque behavior, which can be linked to transition effects or entry- and exit actions. For parameters and properties not only the UML primitive data type Integer can be used, but also user-defined types like enumerations, which can also be defined in the class diagram, or other classes. For parameters of operations and properties of signals it is recommended to use user-defined data types instead of UML primitive data types.

According to a former publication [KSA09] it had been planed to implement both signal receptions and operations for input events as well as for output events. Since operations are supposed to be synchronous, it had been planed to implement the support for return values. An operation in an environment class using a return value would have resulted in an output event directly followed by an input event representing the return value. An operation in a system-under-test class would have resulted in an input event directly followed by an output event representing the return value. However, the feature to use operations in a class of the system-under-test has not been maintained, and the feature to use operations in an environment class still lacks the support for return values in the current version of *UMMU*.

Class diagrams are also used to statically create the instances of the classes. This is done by using so-called “instance specifications”. There also the initial value for the properties can be assigned including references to other objects.

For classes, there is a limited support for inheritance without polymorphism and “late-binding” [KSA09].

Classes can either be used to merely encapsulate data, then they can be passive. Otherwise, they are active and a state-machine diagram has to be created for them. Valid input files contain exactly one state-machine diagram per active class. However, the

[1] `http://www.papyrusuml.org/`

state-machine within this diagram can use orthogonal regions in order to model parallel behavior within one class.

In each orthogonal region there is an initial state. This is the state where the transition from the “Initial” element (the black dot) points to. From this state, there has to be at least one outgoing transition. This can either be a self-loop or a transition to any other state. Transitions can have a trigger, a guard and an effect.

A trigger is linked to an event. If this event occurs, the trigger is said to “fire” and the transition is taken. Currently supported events are the reception of a signal (SignalEvent), the lapse of time (TimeEvent), or an event, that occurs when the evaluation of a user-defined Boolean expression in OCL changes from false to true (ChangeEvent).

A guard is a Boolean expression, which has to be true, in order that the transition can be taken, if the trigger fires. Thus, a guard can hinder a transition to be taken. Additionally to arbitrary expressions in OCL there are two special expressions: “false” and “else” As suggested by the name a “false” guard disables a transition completely, whereas an “else” guard can be used if there are two transitions with the same trigger as a short-hand form of the negation of the guard of the other transition.

In an effect, opaque behavior can be defined to perform a state variable update, a call of an operation or the sending of a signal to other objects. The language used for opaque behavior is called *AGSL* and described in Section 3.1.3. Opaque behavior can also be linked to the entry or the exit of a state, which is useful if there is more than one transition leading to or away from a state and the effect shall happen either way.

3.1.2 Mogentes UML Profile

UML diagrams can only be processed by the toolchain, if the custom “Mogentes UML profile” is applied to it. Using this profile one can add information specific for the fault based test case generation to the model. The following stereotypes are provided by the profile:

«system_under_test» This stereotype is used to mark exactly one class as system-under-test. However, the system-under-test can consist of more than one class. As described in a publication by the developers [KSA09] all classes that are not marked with the sterotype «environment» are considered to be part of the system-under-test.

«environment» This stereotype is used to mark one or more classes as environment. Method calls and signal receptions of these classes are later mapped to observable actions.

«from_environment» This stereotype can be applied to both signals and signal receptions. If it is used to mark a signal, then this signal is broadcasted to all instances of all classes, which have a reception for this signal. If it is used on a signal reception, then the signal is only broadcasted to the instances of the particular class.

«range_definition» This stereotype is used on user-defined data types in order to limit the range of Integer and Float types. Limiting the range is very important for test case generation approaches, which enumerate all possible values as *Ulysses* does.

«range» This stereotype is used to denote that a user-defined range is used for a property.

«sample_definition» When limiting the range of an Integer to a continuous range is not restrictive enough to cope with the complexity, this stereotype can be used. By defining a sample one can limit the values used for enumeration to a definite set of concrete values.

«sample» This stereotype is used to denote that a user-defined sample is used for a property.

«triggerless» In UML a so called change-trigger only fires, when the evaluation of an expression changes from false to true. If a transition has no trigger at all, a transition can only be taken if the guard holds, when the source state of the transition is entered. The stereotype «triggerless» has been introduced as combination of both. A transition having no trigger and a guard when marked with the stereotype «triggerless» is taken whenever the guard evaluates to true.

3.1.3 Activity and Guard Specification Language

The activity guard and specification language (AGSL) is a "programming language embeddable in UML model elements". It has been developed by Budapest University of Technology and Economics and is described in the appendix of a technical report [Kro10].

For the use in the *MoMuT* toolchain, the following statements are supported:

Assignment. An assignment has the form `prop = expression;`, where `prop` is the identifier of a property of the current object and `expression` is an expression.

Invoke Statement. An invoke statement has the form `run op(params);` where `op` is the qualified name of an operation of any object that the current object has a reference to and `params` is a parameter for the operation. Note, that parameters are optional for operations. Operations are used to model outputs from the system-under-test to the environment, if the output is modeled as operation.

Send Statement. An invoke statement has the form `send sig(params) to obj;` where `sig` is the identifier of a signal, `params` are optional parameters and `obj` is the object that shall receive the signal. There are two usages of the send statement. Send statements can be used to model an output by sending a signal to a class belonging to the environment. Send statements can also be used to dispatch inputs received from one class of the system-under-test explicitly to other classes, which also belong to the system-under-test.

3.2 UMMU

The first tool of the *MoMuT* toolchain is called *UMMU*. It translates the UML test models into object-oriented action systems, which are used as intermediate language. Also, here the mutation operators for creating the model mutants are implemented. The output of this tool are both the original model and model mutants encoded as object-oriented action systems. This tool has been developed by AIT within the MOGENTES project.

3.2.1 VIATRA

For parsing, mutating and transforming both original and mutated models into OOAS, the VIATRA Framework is used. VIATRA is an acronym for "Visual Automated Transformations for Formal Verification and Validation of UML Models". It has been developed at Budapest University of Technology and Economics before the MOGENTES project has started [Cse+02].

VIATRA is based on Eclipse. It provides so-called model spaces, in which the test model can be imported and transformed.

3.2.2 UML to OOAS transformation

The general ideas of the transformation and the model mutation have been presented in a publication presenting the mapping [KSA09]. Even though this paper is based on a previous version of the toolchain, most parts are still correct, only the notion of time has changed slightly since.

All classes of the test model that correspond to the system-under-test (and not the environment) are directly mapped to a corresponding class in the object-oriented action system. Additionally there is one so-called "housekeeping" class with the name `__model` and one class called `__environment` in the OOAS, which represents all classes marked as environment in the UML class diagram.

The OOAS class `__model` is marked as `autocons`, thus one object is automatically created and used as root object. In the variable definition block of this class all instances of the system-under-test as defined in the UML class diagram are defined. Additionally one object of the `__environment` class is created.

In the system-assembling-block the prioritized composition is used to give the actions of the `__model` class priority over the system-under-test classes, which are themselves connected with a non-deterministic choice among each other and have priority over the actions in the `__environment` class.

For the test model of the car alarm system, described in Section 1.6 the system-assembling-block looks as follows:

```
__model // ( AlarmSystem ) // __environment
```

The input actions are defined in the `__environment` class. Since this class has the lowest priority in the system-assembling-block, so-called run-to-completion semantics is achieved. That means that new inputs only occur in the test model, when the internal actions caused by the last input have finished.

One important aspect of the input handling is, that in each state of the action system, only those input actions are enabled which correspond to trigger events of transitions which are outgoing from the current state and enabled by the guards. Because of that, the action system is not input-complete. This is very important in order to enable partial models.

When an input-event is chosen, the object of the `__environment` class writes a corresponding event into an event queue. If there is only one object, which uses this event as transition trigger, the queue of this object is used, otherwise it is broadcasted via the housekeeping class, which implements an observer pattern.

The transitions themselves, as well as their effects are translated into several internal actions. For each transitions between two states an internal action is created. In each class for each orthogonal region of the state machine and for each nested state there is a variable storing the current state. The type of each of these variables is an enumeration with all possible states in this region resp. nested state. The action representing the transition consists of a guard with the condition that the source state of the transition equals the current state and the body contains a state update, which updates this variable with the destination state of the transition. If the transition is triggered by an event the guard additionally contains the condition that the event has to be on top of the queue and the body contains a variable update which eliminates this event from the queue.

Opaque behavior of effects or entry and exit actions is also translated into several actions. Each assignment statement is translated into an internal action, calls of and signals to the environment are translated into output actions.

In the so-called **do od**-block, the iterative loop of each class, these actions are composed as follows: for each transition between two states, first the transition, then the actions which correspond to the exit action of the source state, then the actions corresponding to the effects of the transitions and finally the actions corresponding to the entry action of the destination state are connected using the sequential composition. The blocks for different transitions are then composed using the non-deterministic choice.

For time handling, originally there had been a special output event called `after(t)` which had been non-deterministically composed with the input actions in the `__environment` class. The parameter `t` denoted the time that had passed in the same time unit that has been chosen for the UML input model.

In current versions of *UMMU* this action is now internal. In order to make the time visible within the LTS, each other input- and output- action now has one additional parameter. The first parameter of each input- or output action denotes the time that has passed since the last input- or output action. A comparison between these two ways of time handling can be found in Section 5.1.3.

One very important aspect of this notion of time is the complexity of exploring the action system using an enumerative approach as Ulysses does. Each possible value of each parameter of each action leads to a new branch which has to be considered in a breadth-first-search. As one can easily imagine, this can lead to enormous run time problems unless the range of possible values is restricted rigorously. Because of that *UMMU* provides different options for picking only "interesting" time steps. Using the default behavior of generating an OOAS with *UMMU* only time values occur in the LTS, which correspond to a time trigger used by a transition that is reachable by the current state of any orthogonal region.

Example As example for an input event consider the "Close" signal from above mentioned car alarm system. The code that is produced for this input action looks as follows:

```
1  ctr receive_external_signal_Close_at_InstanceSpecification_0(c_wait_time :t_time)
2    = requires m.Close_is_enabled_at_AlarmSystem(0) and
3        ( m.get_wait_time() = c_wait_time ) :
4    # re-allow wait controllable
5     wait_allowed := true;
6    m.reset_wait_time();
7    # reset flags enabling external input
8    m.disable_external_input();
9    m.get_InstanceSpecification_0().__rcv_Close();
10    check_for_inputs := true
11 end;
```

The prefix `ctr` is used to mark the action as input, so that its occurrence leads to an input label in the produced LTS and is therefore considered as input event in the test case. The name of the action is derived from the name of the signal, but the label contains additional information, like the object, that receives the signal (`InstanceSpecification_0`). The parameter denotes the time, that has passed before this action occurs. The guard in lines 2 and 3 consists of a method call to the "housekeeping" object to ensure, that one of the transitions triggered by the reception of this signal is enabled and the parameter denoting the time matches the internal simulated time.

Setting the `wait_allowed` flag to true in line 5, the action re-enables the internal `after` action. The internal simulated time is reseted in line 6, so that the time parameter of the next visible action will show the time, that has passed since this action has occurred. In line 8 the method `disable_external_input()` is called on the housekeeping object. This resets the variables containing the information, which input actions are enabled because they have a corresponding internal action encoding an enabled transition in the state machine. Using the method `m.get_InstanceSpecification_0().__rcv_Close();` in line 9 the event is written into the queue of the object `InstanceSpecification_0`. By setting the flag `check_for_inputs` to true in line 10, in the next iteration of the **do od**-block an internal action gets enabled. This action evaluates, which input actions shall be enabled because their corresponding transitions are.

In the following as example for an internal action encoding the actual transition within the state machine is given. The transition from the state “OpenAndLocked” to the state “ClosedAndLocked” looks like this:

```
trans_OpenAndLocked_From_OpenAndLocked_to_ClosedAndLocked_Trans_ClosedAndLocked =
  requires ((Region_0 = AlarmSystem_Region_0_OpenAndLocked) and
    (not __consumed_Region_0) and (self.__events <> [nil]) and
    (((hd self.__events)[0] = __received_AlarmSystem_Close))) :
  # register timer(s) for entered state: ClosedAndLocked
  __m.register_transition_AlarmSystem(20, self, ClosedAndLocked_TimeEvent_3_Armed
      );
  Region_0 := AlarmSystem_Region_0_ClosedAndLocked;
  # set consumed flags
  __consumed_Region_0 := true;
  __consumed_Region_0__Alarm__Region_0 := true
```

It is an internal action, so it has no prefix. The variable `Region_0` is used to store the current state of the orthogonal region. The transition is enabled if the system is in the state “ClosedAndUnlocked” and the first element of the event queue is the event corresponding to the reception of the “Close” signal. The state variable is updated to the enumeration value corresponding to the “ClosedAndLocked” state and since there is a transition triggered by a time trigger leading away from the target state, a timer is registered. For each orthogonal region and for each super-state there is a `consumed` flag to ensure that in each region only one transition is triggered by the event.

An example for an output event is the call of the operation “SetOn” in the environment class “AlarmArmed”. In the OOAS this is translated as output event of the class “AlarmSystem”.

The action looks like this:

```
obs __call_AlarmArmed_SetOn(c_wait_time : t_time) =
  requires __m.get_wait_time() = c_wait_time : __m.reset_wait_time() end;
```

Again, the parameter denotes the time that has passed, before this action occurs. The guard filters the possible time values and only allows the one matching the internal simulated time.

Since this output is produced as entry action of the “Armed” state, this action is coupled to each transition leading to this state. This is done by using the sequential composition within the **do od**-block:

```
#...
[]
  (
    trans_ClosedAndLocked_From_ClosedAndLocked_to_Armed_Trans_0_Armed;
    (var __t26 : t_time : __call_AlarmArmed_SetOn( __t26))
  )
[]
#...
```

The temporary variable `__t26` is used to enumerate all possible time values. There is an implicit non-deterministic choice among each of these values and the guard within the output action filters it.

3.2.3 Model Mutation

The effectiveness and efficiency of a mutation based technique depends on the quality of the mutation operators. By choosing, which mutation operator is applied, it is determined, which possible errors can be predicted. The generated test cases can then be used to decide, whether these errors occur in the system-under-test.

A survey of planned and realized mutation operators within the MOGENTES project has been published in a project deliverable [Wei08].

In the version of the toolchain used for this thesis, the following mutation operators have been implemented. Note, that each mutant has only one deviation from the original. So if an operator replaces something "for each transition", then for each transition a new mutant is created. The rationale behind this is to rely on the coupling effect which described in Section 1.2.2 and create only first-order-mutants.

Removing Trigger Events on all Transitions. This mutation operator removes the trigger of each transition and applies the «triggerless» stereotype instead.

Mutating Transition Signal Events. This mutation operator replaces the signal used as trigger with any other possible signal. Thus, if a class has several signal receptions, due to combinatorial effects this mutation operator creates a high number of mutants. Also this mutation operator is likely to create non-deterministic mutants out of deterministic test models.

Mutating Transition Time Trigger Events. This mutation operator replaces the time trigger expression with six other values: the expression is increased and decreased by one unit (+1, -1), increased and decreased by one order of magnitude (* 10, / 10) and replaced by the highest used value and by the constant value 1.

Mutating Transition OCL Expressions. This mutation operator is responsible for mutating expressions used in guards, which are written in OCL. Every literal Integer value used in a guard is replaced by a value increased by one, a value decreased by one, and the constant value 0. Enumeration literals used in guards are replaced by another value of the same type.

Mutating Transition AGSL Expressions. This mutation operator replaces literal Integer and enumeration values used in assignment statements of transition effects. Integer values are increased and decreased by one and replaced by the constant value 0, enumeration values are replaced by an other value of the same type.

Mutating Guards on all Transitions. This mutation operator mutates guards as whole. On each transition the guard is inverted by negating the condition. Additionally for each transition with a guard the condition is set to logical false and true. Setting the guard to false disables the transition, while setting the guard to true removes the guard.

Mutating OCL Change Expression on all Transitions. This mutation operator inverts the expressions of change triggers.

Removing Entry and Exit Action in all States. This mutation operator removes entry- and exit actions as a whole. If both entry- and exit actions exist in a state, then two separate mutants are created.

Mutating Entry and Exit Action in all States. This mutation operator replaces the entry action of a state with the entry action of another state and the exit action of a state with the exit action of an other state.

Removing Effect in all Transitions. This mutation operator removes the effect of a transition as a whole.

Mutating Effect in all Transitions. This mutation operator replaces the effect of a transition with the effect of each other transition.

Mutating Sub Expressions in OCL Expressions. This mutation operator decomposes the guards and replaces each Boolean subexpression with the negated expression as well as to true and to false.

Mutating Operator in OCL Expressions. This mutation operator replaces an "equal", "unequal", "less", "greater", "less or equal" or "greater or equal" operator with each other operator except for the logical negation. Also it replaces an arithmetic operator with each other arithmetic operator.

Mutating AGSL Expressions. This mutation operator removes a single statement of an opaque behavior.

Figure 3.2 illustrates five mutation operators ("Mutating Transition Signal Events", "Mutating Transition Time Triggers", "Removing Trigger Events", Mutating Guards, and Removing Entry Action) by showing a mutant of each.

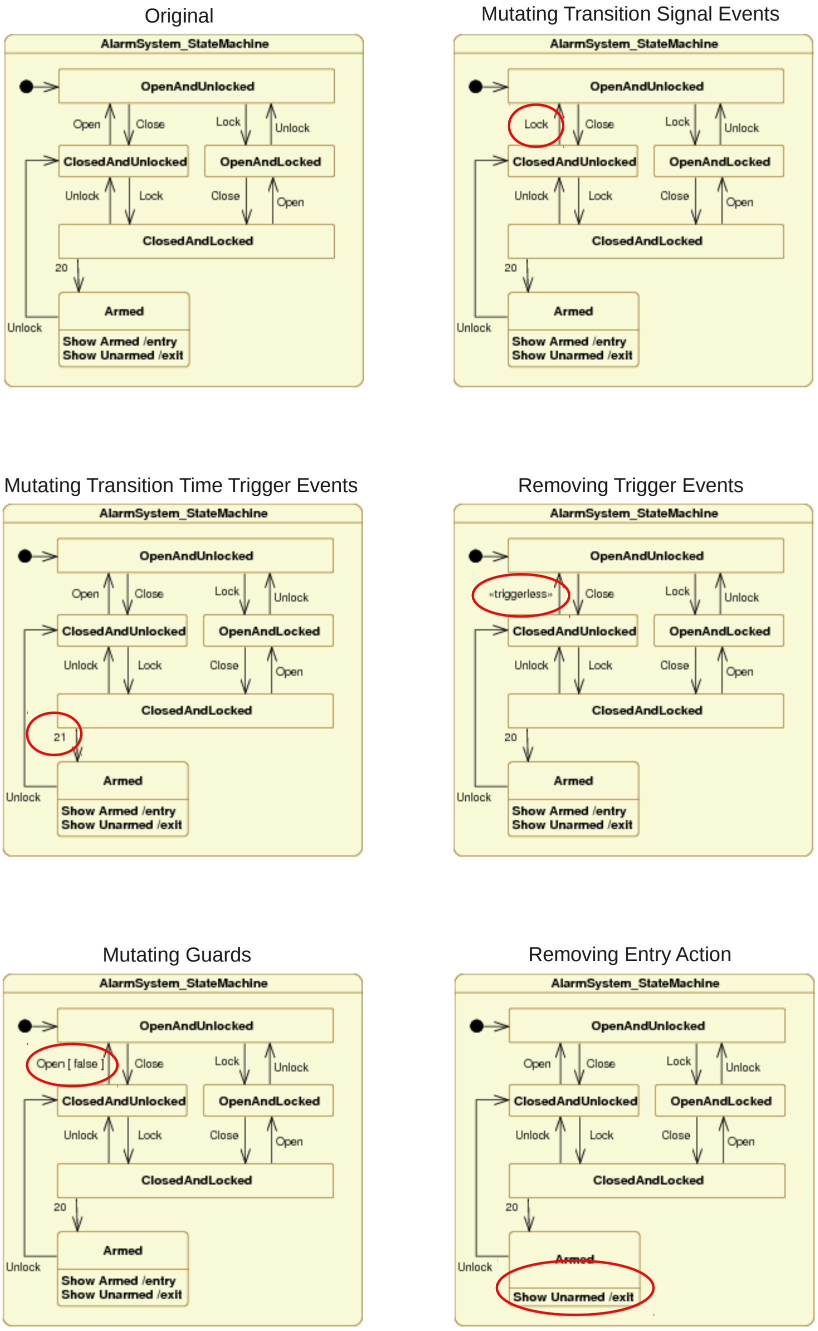
3.3 Argos

Argos is a compiler for object-oriented action systems (OOAS) and has been originally written by Willibald Krenn within the MOGENTES project. It produces flat action systems which are encoded in Prolog files that can be processed by *Ulysses*. Since July 2010 Argos is maintained by the author of this thesis. As a part of this maintenance work, an Argos manual which also describes in detail the concrete syntax of OOAS has been written. [Tir12]

In order to convert the object-oriented action system into a flat one, *Argos* duplicates each variable, method and action defined in a class for each object. This is possible, since objects can only be created in the variable definition block and are therefore known at compile-time. Details on the translation process can be found in the paper [KSA09] describing the approach.

3.4 Ulysses

Ulysses is an *ioco* checker developed by Harald Brandl within the MOGENTES project.



Original

Mutating Transition Signal Events

Mutating Transition Time Trigger Events

Removing Trigger Events

Mutating Guards

Removing Entry Action

Figure 3.2: Illustration of mutation operators

3.4.1 Features

The main usage of *Ulysses* is to check the *ioco* conformance between two action systems. When providing two action systems, one marked as original and one marked as mutant, it can interpret the action systems and check whether the LTS corresponding to the mutated action system conforms to the LTS of the original action system in terms of *ioco*. If it does not, a test case revealing the failure can be generated. When *Ulysses* is used this way, the LTS is constructed on the fly and only used internally. The construction of the LTS is performed by calculating the product graph as described in Section 2.2.2.

Another possible usage of *Ulysses* is the generation of the product graph $as \times_{ioco} asm$ of two action systems as and asm and explicitly store it as file. For doing so one can choose among two different file formats: the "aut" format and the graphviz format. The "aut" format is the same format as used for test cases. Files in this format can also be used as input files, while the graphviz format is for visualization and debugging only.

When calculating the product graph of an action system as with itself $as \times_{ioco} as$, the result equals the LTS of the action system. This means, that calculating the product graph can be used to interpret an action system and store its LTS explicitly as graph.

Instead of providing two action systems, it is also possible to provide one action system and a LTS in Aldebaran "aut" format. This enables the *ioco* check between action systems which have to be adapted first, in order to use the same alphabet. Action systems using different alphabets can arise, when they are produced from different UML models, since in each translation step of the toolchain some information about the internal structure is encoded into the action names, on which the labels of the LTS are based on. Even if the test interface of two UML models is the same, the labels in the produced LTS may differ. Since the "aut" format is human readable, a text editor can be used to relabel the actions of a LTS.

Finally, *Ulysses* can also be used to check, whether a given test case is able to reveal the difference between an action system marked as original and an action system marked as mutant.

3.4.2 Killing Strategies

To determine the conformance between two action systems, *Ulysses* calculates the corresponding product graph. If non-conformance can be shown, at one point there is an output produced by the model mutant, which is not allowed by the original model. In the product graph the last state before this output occurs, is called "unsafe state". From this state there is a transition labeled with the unspecified output leading to a so-called "fail state". This state has a self-loop labeled with "fail".

Ulysses is designed to create positive test cases, which explicitly contain a pass verdict. So a test case can be obtained by expanding a trace, that leads to an unsafe state, by adding only outgoing transitions with an output label defined by the specification representing an allowed output. The output leading to the fail state per definition is not an allowed output and therefore not part of the test case. Instead, after each output passing the unsafe state without reaching the fail-state, the pass verdict is given.

The rationale behind this is, that any implementation, that produces a correct output, does not implement the failure, which had been injected to the mutant by the mutation operator.

Since a failure in a model can cause non-conforming behavior in more than one state, it can happen, that the product-graph contains more than one “unsafe state” and often there is more than one trace leading to a specific “unsafe state”.

Because of that a decision has to be made, how to extract test cases out of a product graph. Throughout the development of *Ulysses*, different strategies have been implemented, but not all have been maintained. The first seven strategies including a so-called random walk have been presented in [Aic+11a]. There a comparison among them is presented using the car alarm system. For more complex models, like the wheel loader, these strategies had been adapted.

The first decision to make is whether test cases have to be linear or if they may be adaptive. Traditionally, test cases had been linear. Since models can be non-deterministic, there can be states, in which there is a non-deterministic choice between two or more output actions, or even states, in which both input- and output-actions can occure. In a linear test case, only one case can be considered per test case. When executing the test cases on an actual implementation, it can happen, that the implementation behaves not as assumed by the test case but still correct with respect to the specification. Because of that, so-called “inconclusive” verdicts have to be added, whenever there is an output in the product graph, that leads away from the considered path to the unsafe state.

A more sophisticated kind of test cases are so-called “branching adaptive test cases”. As the name suggests this kind of test cases allow to represent the branching behavior of the specification in the test case. While linear test cases consider just one path to the unsafe state and stop immediately when this path is left, adaptive test cases can represent any number of paths at the same time. These test cases are general graphs and can even be cyclic. Only if there is a path, from which the *unsafe* state can not be reached at all, an “inconclusive” verdict is given. An example for a cyclic test case is given in Section 5.3.2.

Early versions of *Ulysses* implemented algorithms to create both linear and adaptive test cases. In a previous paper [Aic+11a] a comparison between the seven strategies provided by *Ulysses* and one approach using another model-based testing tool had been presented. Approach A1 and A2 included algorithms which created linear test cases. However, they had severe disadventages like producing far too many duplicate test cases. Thus, they had been given up quite soon.

Current versions of *Ulysses* provide the following of the original presented test case selection strategies:

S3 , also referred to as “Adaptive Killing Strategy”, creates one test case for each unsafe state of the product graph between the original model and a model mutant. The generated test cases are adaptive.

S4 , also referred to as “Lazy Killing Strategy”, also considers each unsafe state. However, before creating a new test case, it checks whether a given unsafe state is already covered by an existing test case. Only for unsafe states, that have not been covered, new test cases are generated.

S5 , also referred to as “Lazy Ignorant Killing Strategy”, is even more restrictive about creating new test cases. Before calculating the product graph between original model and model mutant, an *ioco* check between each existing test case and the model mutant is performed with the test case as specification and the model mutant as implementation. If non-conformance is shown, the test case is said to kill the mutant. In this case processing the entire mutant is skipped and no additional test case is produced. However, if no pre-existing test case is able to kill the model mutant, for each unsafe state of the product graph a test case is created.

S6 , also referred to as “Random First Killing Strategy”, is very similar to S5, with the only difference, that it starts with an initial test suite. This initial test suite consists of test cases gained by so-called random walks. A random walk is a random path through the LTS of the original model.

S7 , also reffered to as “Random Killing Strategy”, consists solely of random walks, without any fault-based approach.

In each fault-based strategy the creation of the product graph is bounded by a user-defined search depth. For small models like most of the models of the car alarm system this is more or less irrelevant, since the models are circular and it is feasible to use a search depth, which is higher than the length of the longest path. For larger models, like in the wheel loader case study, this is different for two reasons: firstly, the run-time of the creation of the product graph grows exponentially with the number of different possible actions, which are possible in each step. So if a high search depth is needed, the process becomes computational infeasible. Secondly, a fault in the model can lead to numerous unsafe states in the product graph. The reason for this is, that non-conforming behaviour of the model mutant can occure in more than one branch. Even though the exploration stops in a branch, as soon, as the first fail state is reached, other branches might be explored far deeper. So even if a fault in a mutant manifests itself in a fail-state that is found early in the generation, exploring alternatives branches might be necessary. Besides the long run-times for complex models this also results in huge test suites, which in general is also considered as undesirable.

For this reason, current versions of *Ulysses* provide an option to alter the above-mentioned test case extraction strategies as follows: instead of exploring the model always up to the given search depth, if the option is enabled, *Ulysses* now also stops at the depth of the first unsafe state. So if an unsafe state is found, the breadth-first-search only finishes the current level but does not go any deeper. The rationale behind this optimization is the reduction of both size of the test suite and test case generation time. Note, that already in the original strategies *S5* and *S6* not all unsafe states are covered. If a mutant can

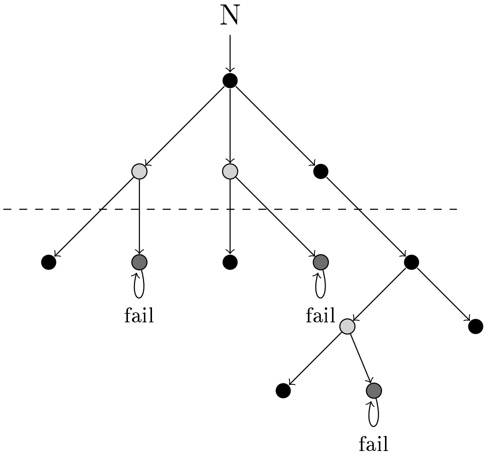


Figure 3.3: The cut of the search tree as product graph

already be killed by an existing test case, which only means that at least one unsafe state is covered, no further test case is generated.

Figure 3.3 visualizes the effects of the altered killing strategies using the product graph. The product graph serves as “search tree”, even though it is not necessarily a tree, since nodes are merged, if they represent an identical state. For creating new test cases, this graph is searched for *unsafe* states. This is done by searching for *fail* states, because an*unsafe* state is a parent of a *fail* state.

When the altered version of a killing strategy is applied, the search stops at the dashed line. However, in other branches of the search tree *unsafe* states can occur in higher search depths, visualized as below the dashed line. For these *unsafe* states no test case is generated.

This option is referred to as “Search Tree Cut” in the rest of this thesis.

4 Combining Model-Based Mutation Testing with Refinement

4.1 Partial and Underspecified Models with ioco

Partial models are one way to handle the complexity that arises with large systems. The idea is to split the system into functional components. Each partial model concentrates on one functional component and leaves other aspects underspecified.

When building abstract models that shall conform to the full model in terms of *ioco*, two cases have to be distinct: in the first case, the underspecification regards input behavior. This can be expressed implicitly, since *ioco* is designed to not care about undefined inputs. Within this thesis, this concept is referred to as “partial models”. In the second case, the underspecification regards output behavior. This has to be expressed explicitly by using a so-called non-deterministic choice.

Figure 4.1 shows an example demonstrating the decomposition of a model into partial models using additional input behavior. The models are presented as LTS. As introduced in Section 2.3.1 the prefix “ctr” is used to denote input labels and the prefix “obs” is used to denote output labels. Note, that the labels correspond to the actions of the underlying action systems, but the labels have been simplified for legibility reasons. Parameters are also part of the label and denote the time that passes before the corresponding action occurs.

The system F is a simplified model of the car alarm system. After the car is closed (action *Close*) and locked (action *Lock*), the alarm system goes into an armed state. (action *ArmedOn*). The delay of 20 seconds is denoted in the parameter. Now two things can happen. Either an authorized person unlocks the door (action *Unlock*), which turns off the arming of the alarm system, or an unauthorized person opens the locked door by force which turns on the alarm.

Since *Open* and *Unlock* are both input actions, their common source state is a good choice for decomposing the system into partial models. P_1 only models the behavior of an authorized unlocking, while P_2 only models the behavior of an unauthorized opening of the locked car.

Figure 4.2 shows an example demonstrating underspecified output behavior, which can be used to express non-determinism. The abstract model A is again inspired by the car alarm system. It defines, that after an input action *Open* has been received, the system has to react with two output actions, *SoundOn* and *FlashOn* but the exact order

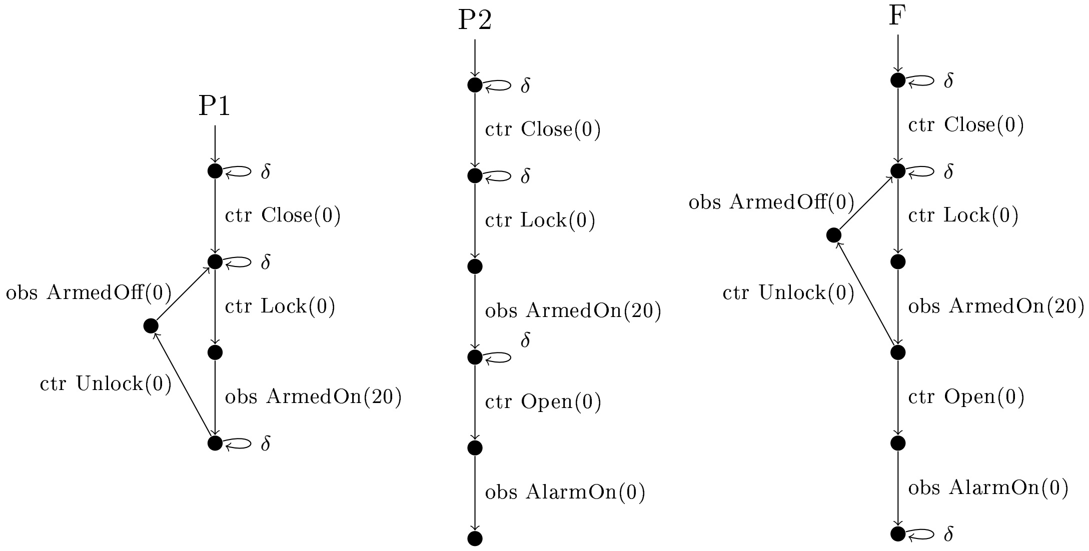


Figure 4.1: Partial models using implicit ambiguity of input behavior

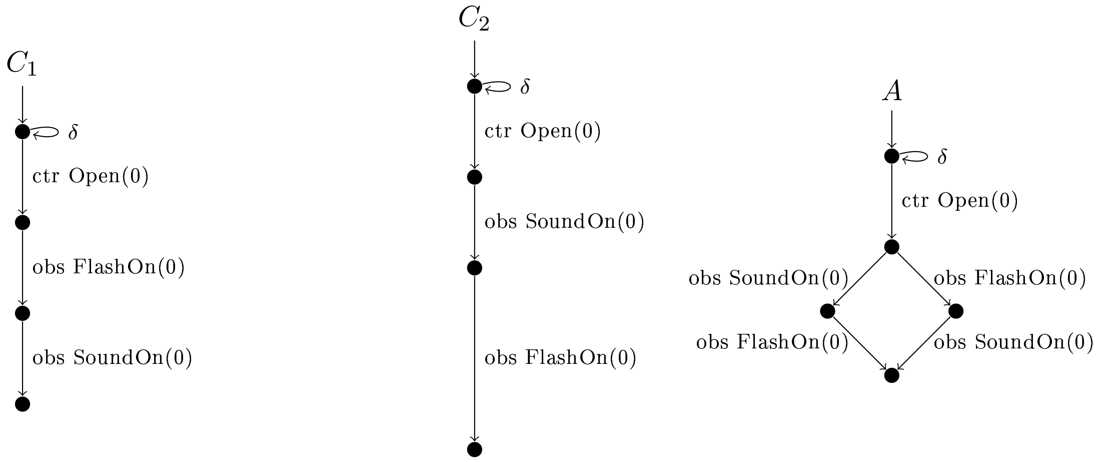


Figure 4.2: Underspecified abstract model A allowing two different concrete implementation models C_1 and C_2

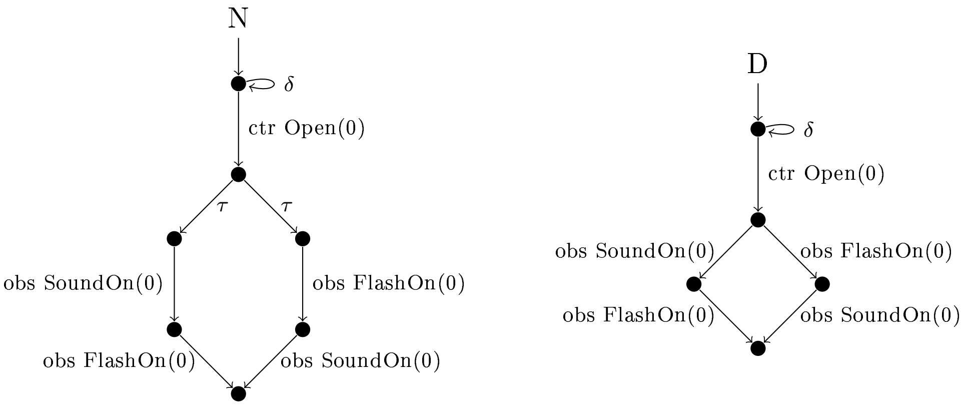


Figure 4.3: Non-deterministic expressed by a non-deterministic LTS and its determinized version

is unspecified. This is expressed by explicitly drawing both possible execution paths, which is also called “non-deterministic choice”. C_1 and C_2 are both models, which are concrete about the specific order and are both conforming in terms of *ioco* to the abstract model A.

This kind of underspecification is often used when modeling processes which run in parallel on the system-under-test. In this case the scheduling of processes can lead to many possible interleavings, which causes the specific order of the events to become unpredictable.

Note, that the term *non-deterministic choice* refers to the specified behavior and not the structure of the LTS. In fact, a *non-deterministic choice* can be expressed either by a non-deterministic LTS using internal transitions, or by deterministic LTS, as shown in Figure 4.3. Both LTS model a system, which after receiving the input *ctr Open(0)* produced the outputs *obs SoundOn(0)* and *obs FlashOn(0)* in either order. In the LTS N, the internal choice between the two possible interleavings is explicitly expressed using two internal τ transitions. Since there is a state, from which there are two outgoing transitions with the same label, it is a non-deterministic LTS. The LTS D is the determinized version of N, where all internal transitions are hidden. It is deterministic, since there is no state, with two transitions having the same label.

A special case of this kind of non-deterministic behavior are so-called mixed states. In a mixed-state, both input and output actions can occur. Figure 4.4 shows an example LTS and its determinized suspension automaton. The example shows again a part of the car alarm system: 20 seconds after the car is closed and locked the alarm system goes into an armed state. However, if the car is unlocked or opened before that 20 seconds have passed, the system stays unarmed. The passing of time itself is not observable, thus it is represented as internal action τ. When creating the suspension automaton, the internal actions are deleted and there is a state, from which there are outgoing transitions with both input and output labels.

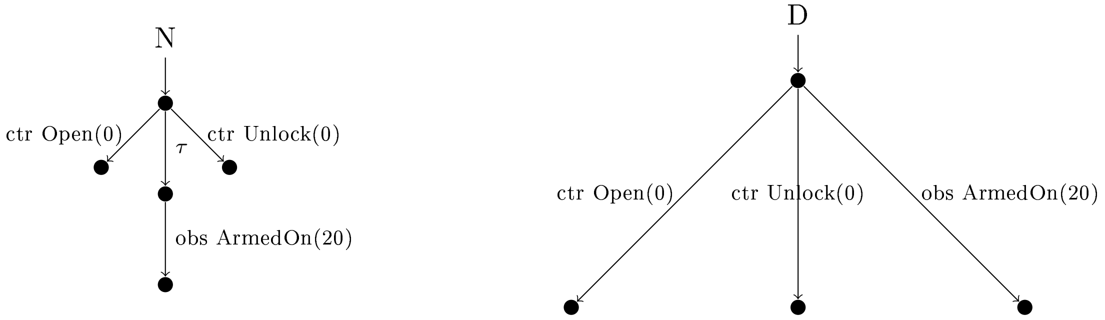


Figure 4.4: Mixed state expressed by a non-deterministic LTS and its determinized version

4.2 Model Refinement

The idea to start developing a system by initially creating an abstract model, which is later refined, is quite old. One of the first methods, using this approach is the Vienna Development Method (VDM) [Jon90]. There the refinement steps consisted mostly of transforming mathematical abstract data types into data types that can be used in implementations. This technique is often referred to as “data refinement” but sometimes literature [Jon90] prefers the term “data reification” instead of refinement.

In general, literature [Geo+95] distinguishes between two different types of refinement:

1. program transformation
2. invent and verify

The idea of program transformation is to apply transformation rules which are guaranteed to produce new models that are refinements of the original ones. The idea of invent and verify is to generate new models independently of the original ones and verify the refinement afterwards. The most well-known technique to do program transformation for refinement is called “refinement calculus” [BVW98]. Examples for the invent and verify method are VDM, the B method [Abr96] and its successor Event-B [Abr10] or RAISE [Geo+95]. In this thesis refinement refers to the *invent and verify* approach.

Another important aspect is to distinguish horizontal modularity and vertical modularity. Beginning with an abstract model of the system, which later is refined step-wise, usually is referred to as vertical modularity. Horizontal modularity on the other hand refers to the decomposition of a system into functional components. In this thesis both types of modularity are considered, but only the vertical modularity is referred to as refinement, while horizontal modularity is referred to as partial models.

One concept of refinement that is related to the refinement based on *ioco* of our approach is the so-called “traces refinement”. It is one of three refinement relations for Hoare’s Communicating Sequential Processes (CSP) [Hoa78], which are implemented by a refinement checker called “FDR2”.

It is defined as:

$$P \sqsubseteq_T Q =_{df} traces(Q) \subseteq traces(P)$$

It means that a process Q refines a process P, if all traces of Q are possible in P. [FSE10]

Note, that the special process *STOP*, which does not perform any events at all, but deadlocks immediately is a traces refinement of any other process. Because of that, there are two additional refinement relations called “Failures refinement” and “Failures-Divergences refinement”. Failures refinement additionally insists, that the refining process does not refuse more events than the refined process and a deadlocked process refuses every event. Failures-Divergence refinement additionally takes care of so-called livelocks. Livelocks are states, in which a process performs an infinite sequence of internal events.

Note, that using partial models, as described in Section 4.1 would not work with above mentioned refinement relations. Instead, already the abstract model has to be complete. The reason, why partial models can be used as specification in *ioco*, is that *ioco* only considers traces, which are defined by the specification.

Since *Ulysses* is an *ioco* checker, it can be used in order to support refinement steps, using *ioco* as refinement relation. It can be determined automatically, if a more specific model refines an abstract model up to a certain depth. Since *ioco* is used, the refined model can be more partial than the refining model.

4.3 Test-Driven Development Process

In a technical report [ALT12b] the current workgroup of Prof. Aichernig (including the author of this thesis) has proposed a new test-driven development process based on model-based mutation testing. Test-driven development has been made popular by Kent Beck [Bec02]. The idea is to start the development of a system by writing test cases, which fail at the beginning. Then the system is implemented so that the test cases pass. Traditionally concrete test cases are written and unit testing frameworks are used for this approach.

However, we suggest to use model-based testing techniques. At first this idea might sound counter-intuitive, since this requires one to build a test-model first, which is used to generate abstract test cases. To run these test cases on the implementation under development, additionally a test adapter has to be written. On a second thought however, this can also be seen as big advantage. If requirements change, only the test model and the test adapter have to be adjusted. There is no need for manually updating the test cases.

Another advantage of using models to generate test cases is that models can be verified to fulfil expected properties. Even if there is no support for model checking for the chosen input language of the test model, a representative set of generated test cases can be used as abstract representation of the model. This idea is not subject of this thesis and has been discussed in detail in another publication [ALT12a].

One main idea of this proposed development process is to use refinement techniques. In early development phases a series of partial test models in terms of horizontal modularity shall be created. That means that each of this models shall capture a different part of the functionality. In later development phases, refinement steps in terms of vertical modularity can be taken, for example by merging some of the vertically modular partial test models. These steps can be supported by conformance checking, which validates the refinement steps.

The development process as shown in Figure 4.5 consists of several steps, divided into two phases: an initial phase and an iterative phase.

In the initial phase the test interface has to be defined. When using the *MoMuT* toolchain, this means for the test model to define the signals and operations, which correspond to the inputs and outputs and therefor the stimuli and responses from the system as described in Section 3.1.1. Then, the test interface has to be defined on the implementation level. This means, that for each input defined by the test model, a concrete stimulus has to be provided and also the responses from the system have to be defined. How this is done depends highly on the used programming language, framework or user interface that is used. For instance this can be done, by defining methods for the stimuli and call-back methods for the responses. Another possibility would be to define input commands and output messages for a command-line interface.

Next, a test driver and test adapter have to be implemented. Here the abstract test cases are parsed and executed on the system. The abstract inputs have to be mapped to the concrete stimuli, which are sent to the system-under-test and the concrete responses have to be received, a look-up for the abstract outputs has to be performed and the resulting output has to be compared to the expected ones. Here it has to be taken care of the structure of the test cases and the underlying testing theory. For the *MoMuT* toolchain this means for example that the test cases are adaptive, having a tree structure and that they are positive test cases having only the *pass* and *inconclusive* verdict explicitly contained, while *fail* verdicts are implicit.

As soon as test driver and test adapter are available, the iterative phase of the process begins. A first partial model is created and abstract test cases are generated. Then the system is implemented and refactored until the test cases pass. In a next iteration a new partial model is built, either capturing different functionality leading to a horizontal step, or being the refinement of one or more existing partial models, leading to a vertical refinement step.

Figure 4.5 sketches, how a series of partial models of a system under development could look like. In this schematic view, for example initially three partial models P_1, P_2, and P_3 are created and test cases are generated independently. The next partial model P_4 would be a refinement of the former models. This can be proven by running the conformance check with P_4 in the role of the implementation and each other partial model in the role of the specification. If the conformance holds, the test suite generated for P_4 can be initialized with all existing test cases. Then, using a regression based approach, additional test cases are only generated, if the existing test cases would not able to reveal all

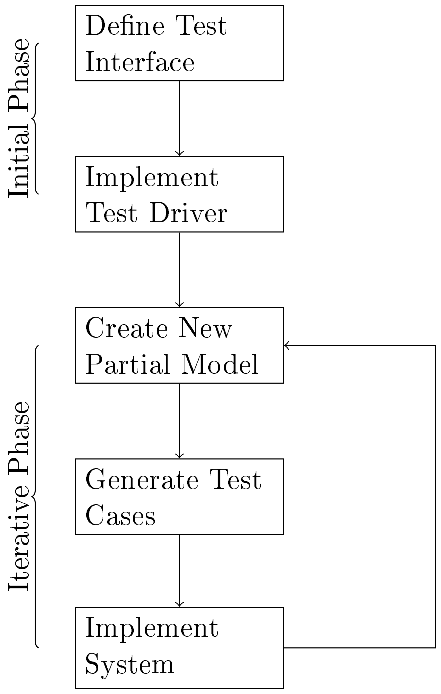


Figure 4.5: Illustration of the development process

mutants of P_4. In a next step another partial model P_5 would be created, in which some new functionality is modelled. It does not have to conform to the prior models, so the test case generation starts again with an empty test suite. In a further step, a new partial model P_6 can be built, which contains the functionality of all prior partial models. Here, again all existing test cases can be used to initialize the test suite for the test case generation.

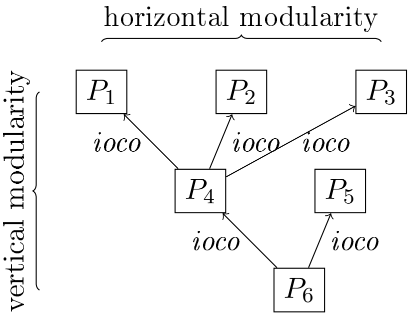


Figure 4.6: Conformance and refinement steps of the partial models

5 Case Study: Car Alarm System

The first case study is based on the car alarm system demonstrator from Ford which is already described in Section 1.6.

The test model depicted there has also been used by Aichernig et al [Aic+11a] to compare a total of 8 different approaches to generate test cases. For this paper a Java implementation has been written by Willibald Krenn. Also, for these former experiments a set of faulty implementations has been derived from this implementation using the mutation tool μJava [MOK05]. These faulty implementations have been manually analyzed, whether they are equivalent to one of the other implementations leading to a total of 38 unique faulty implementations.

Finally, there also exists a test driver that parses abstract test cases in Aldebaran aut format and executes them on either the original implementation or one of the faulty implementations.

Being well-known this demonstrator is very useful in order to show new ideas. In this chapter, decomposing a test model into partial models is shown. Also, the impact of different modeling styles is investigated. For each model test suites are generated. Using classical mutation testing, the given faulty implementations are used to measure the quality of the test suites.

5.1 Common Modeling Details

5.1.1 Refinement

Just like in the technical report [ALT12b], the car alarm system has been split up into two partial models in terms of horizontal modularity. Note, that the author of this thesis, also contributed the models and the experimental results of test case generation to the technical report. There, object-oriented action systems had been used directly as input language. For this thesis, the experiment has been repeated using UML models as input files and therefore the entire *MoMuT* toolchain is used.

Figure 5.1 shows the conformance relation between the two partial models cas_1 and cas_2 and the full model cas_3.

Figures 5.2 and 5.3 illustrate the decomposition into partial models using the UML state machines.

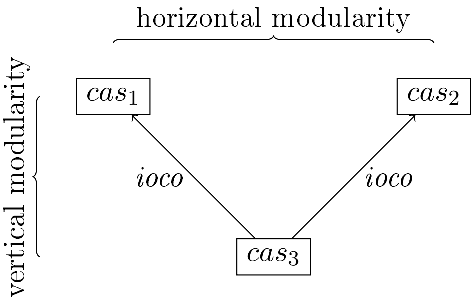


Figure 5.1: Conformance and refinement steps of the partial models (car alarm system)

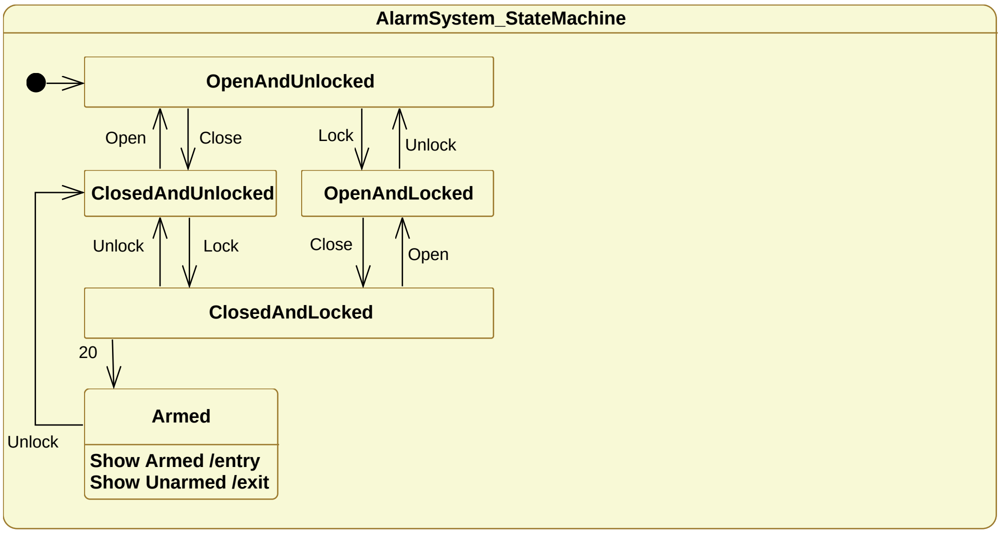


Figure 5.2: State machine of the car alarm system, cas_1

In the model cas_1, depicted in Figure 5.2 only behavior is modeled that leads to the *Armed* state. After reaching the *Armed* state only *Unlock* is defined, which disables the arming.

Partial model cas_2, shown in Figure 5.3 is the counter part. Here, there is only one path from the initial state to the *Armed* state. In this trace first the door is closed, than the car is locked and finally 20 seconds are waited. However, for cases where an intrusion has been detected, all alarm behavior is modeled.

5.1.2 Non-Determinism

While the original UML model, which was created within the MOGENTES project, represented a deterministic system, the OOAS model used in the technical report [ALT12b] represents a non-deterministic system.

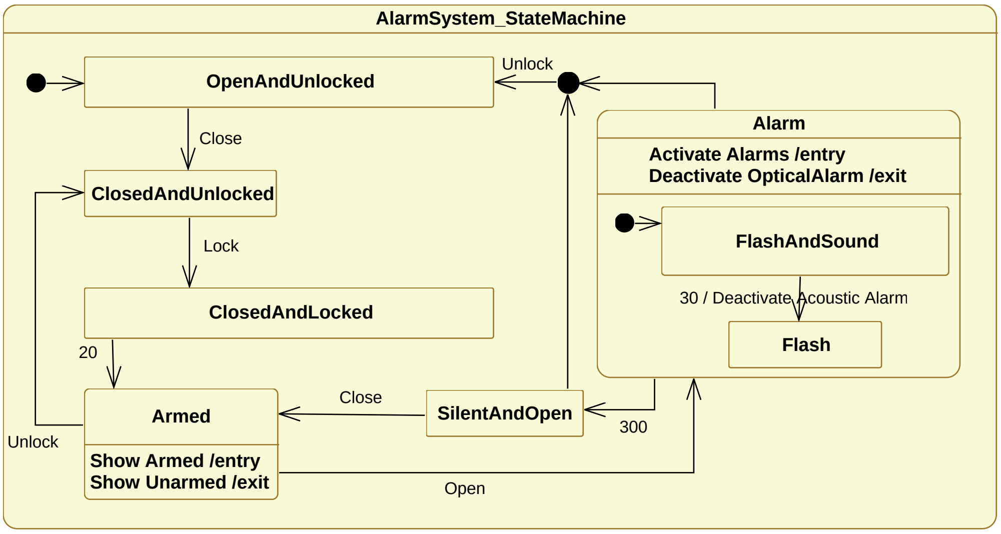


Figure 5.3: State machine of the car alarm system, cas_2

Non-determinism in terms of underspecification is introduced to model parallel behavior. For instance, there are situations, in which two events shall happen at the same time: when both flash and sound alarm are switched on or off, the original model prescribes a specific order. Since our toolchain supports non-determinism, this is not necessary. Instead, the non-determinism can be used to model both possible interleavings.

5.1.3 Time Handling

One important aspect of modeling the car alarm system is the lapse of time. Two out of the three core requirements prescribe timing behavior. In UML this is expressed by so called time triggers. Time Triggers are events, parametrized by an Integer value denoting the time in a fixed unit. OOAS on the other hand lacks the notion of time. Within the MOGENTES project two possible ways of modeling the lapse of time in action systems have been investigated.

The first notion of time has been the introduction of a new observable action called *after*. This had been inspired by the representation of quiescence in *ioco*, which is also treated as observable action. This worked fine for the car alarm system, because here, each time events are always followed by another observable action.

Figure 5.4 shows a labeled transition system of the deterministic version of the car alarm system using the old notion of time.

The disadvantage of this notion is, that it is impossible to express the difference, whether the delay is controlled by the system-under-test or by the environment. Especially with mixed-states, which can occur in LTS, this distinction can be important.

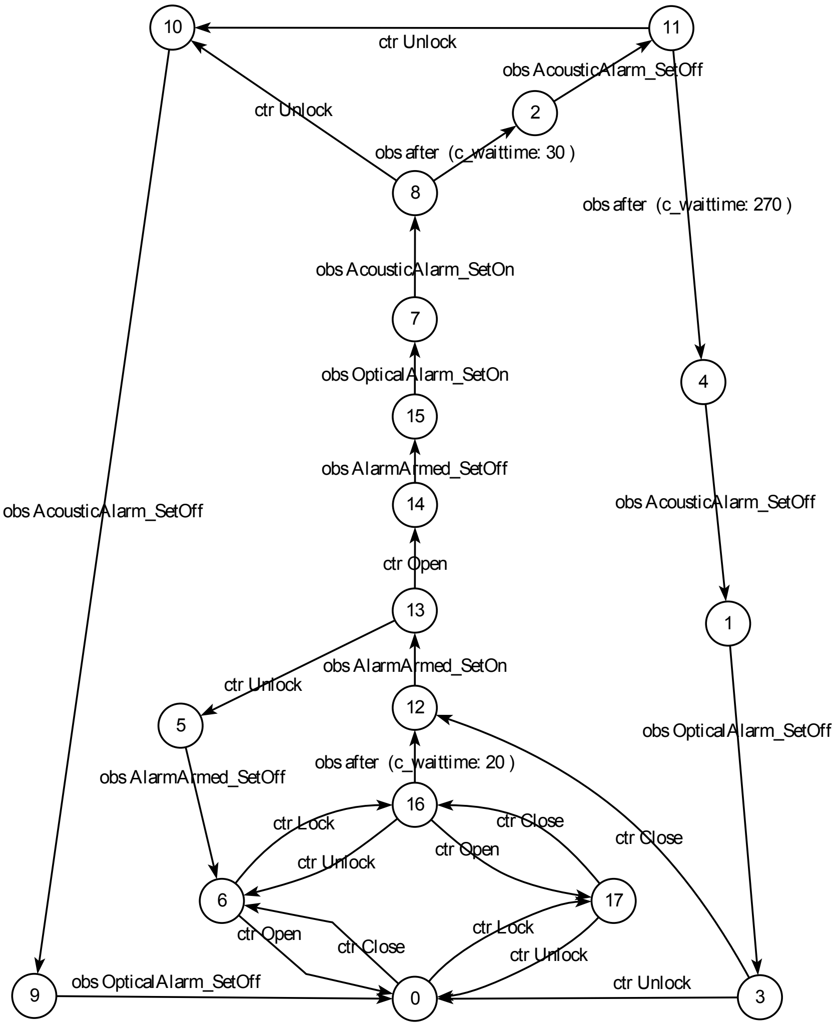


Figure 5.4: LTS of the car alarm system (deterministic, old notion of time)

To overcome this problem a second notion has been developed. In this notion all controllable and observable actions are parametrized with an Integer denoting the time. In the context of a test case, where the tester has the role of the environment, the semantics for these parameters is: if the action is controllable, the tester shall wait the specific time before processing the action. If the action is observable, the occurrence of the action is expected exactly after waiting the specific time.

Figure 5.5 shows a labeled transition system of the non-deterministic version of the car alarm system using the new notion of time.

5.2 Alternative UML Modeling Styles

In addition to the original model, which has been created by AIT and models a deterministic system, within this thesis different non-deterministic versions of the model have been created. As suggested in Section 3.1.1 the variety of supported UML elements, enables different possible modeling styles. In this section three possible ways of modeling the car alarm system are presented:

- Simple State Machines
- Action System Like
- Multiple Classes

They differ in size and complexity, follow different paradigms and processing them using the described tool chain leads to different test suites. They are not meant as suggestion how one should model a system, but they are rather extreme border cases built to demonstrate what is possible and how it effects the generated test suites.

5.2.1 Simple State Machine Modeling Style

The idea behind this modeling style is to directly encode the structure of the intended labeled transition system into one UML State Machine of one active SUT class. From a structural point of view the difference to the original model, which is presented in Section 1.6, is that all outputs to the actuators are put in one environment class. In the state machine diagrams there are two main differences compared to the original model: firstly no sub state is used, but rather all possible states are enumerated explicitly. Secondly, instead of using entry and exit actions in order to model the outputs behavior, here every output has its an own transition.

Figure 5.6 shows the first partial model using this modeling style. Again, the model only contains scenarios up to the point, where the car alarm system is armed. “State_0” is the initial state and corresponds to the “OpenAndUnlocked” state. The states “State_1” .. “State_4” also have a corresponding state in the original model. “State_5” on the other hand is introduced as intermediate step replacing the exit action of the “Armed” state.

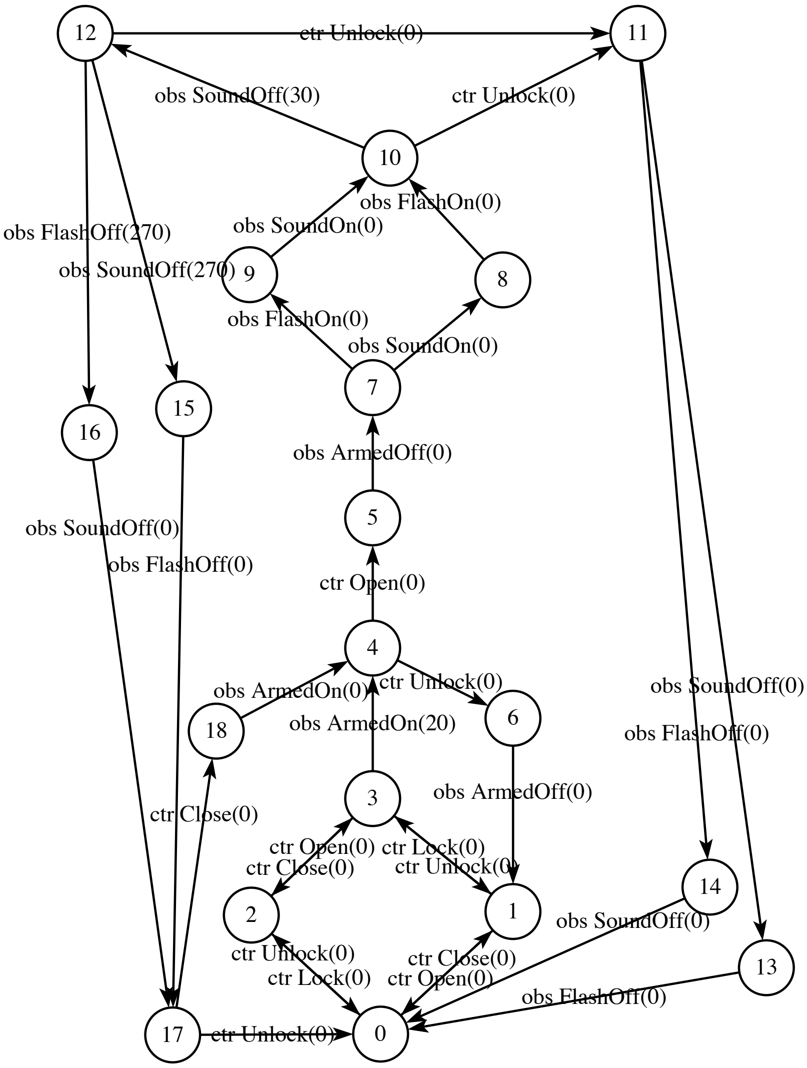


Figure 5.5: LTS of the car alarm system (non-deterministic, new notion of time)

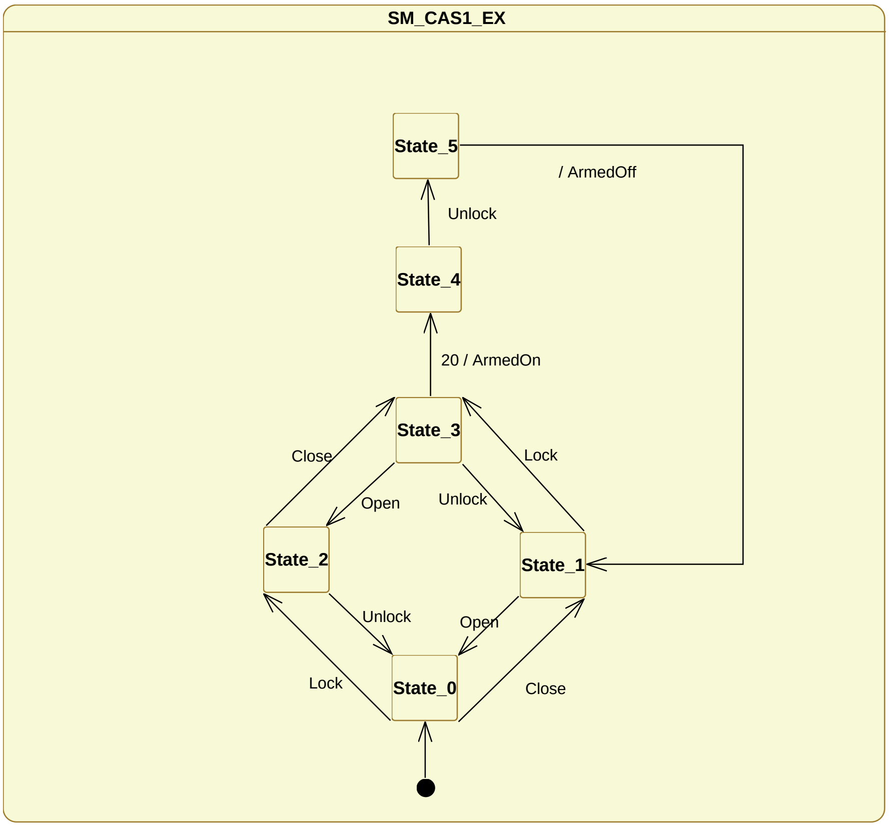


Figure 5.6: State machine, simple state machine version, cas_1

Figure 5.7 shows the complementing partial model using this modeling style. Again, only one path from the initial state to “State_4” which corresponds to the “Armed” state is defined. From “State_6” there are two outgoing transitions with an effect but without a trigger. This is one way to model under-specified behavior. In order to explicitly allow that the output events “FlashOn” and “SoundOn” can occur in both possible orders, this diamond shaped structure is used. A similar structure can be found in the outgoing transitions from “State_10” and “State_11”, where the alarms are turned off in any order. “State_10” corresponds to the case, where the car is unlocked by an authorized person, “State_11” corresponds to the case, where the alarms are turned off after five minutes. Note, that the sound alarm is turned off a second time, in order to be compatible with the existing Java implementation of the car alarm system.

Figure 5.7 shows the full model which combines the behavior of the two partial models.

When translating this model into OOAS using the same translation parameters (ie. Time Handling) like the original model, it can be proven that the deterministic model conforms to this non-deterministic model in terms of *ioco*, but the non-deterministic model does not conform to the deterministic one. The reason for this is, that being in the state, where the alarm is turned on, the non-deterministic allows both orders in which SoundAlarm and FlashAlarm are turned on, whereas the deterministic model prescribes the order precisely. Since for conforming in terms of *ioco* the set of outputs of the implementation has to be a subset of the outputs of the specification, this implies non-conformance. When using *Ulysses* to check this property, one has to make sure, that both models use the exact same labels for input and output actions. This is not the case when using the files generated using *Argos* since a lot of structural information is encoded into the labels of the actions. Practically this can be worked around by storing the LTS into a file in Aldebaran “aut” format and using a text editor to replace the labels into the exact labels of the other action system.

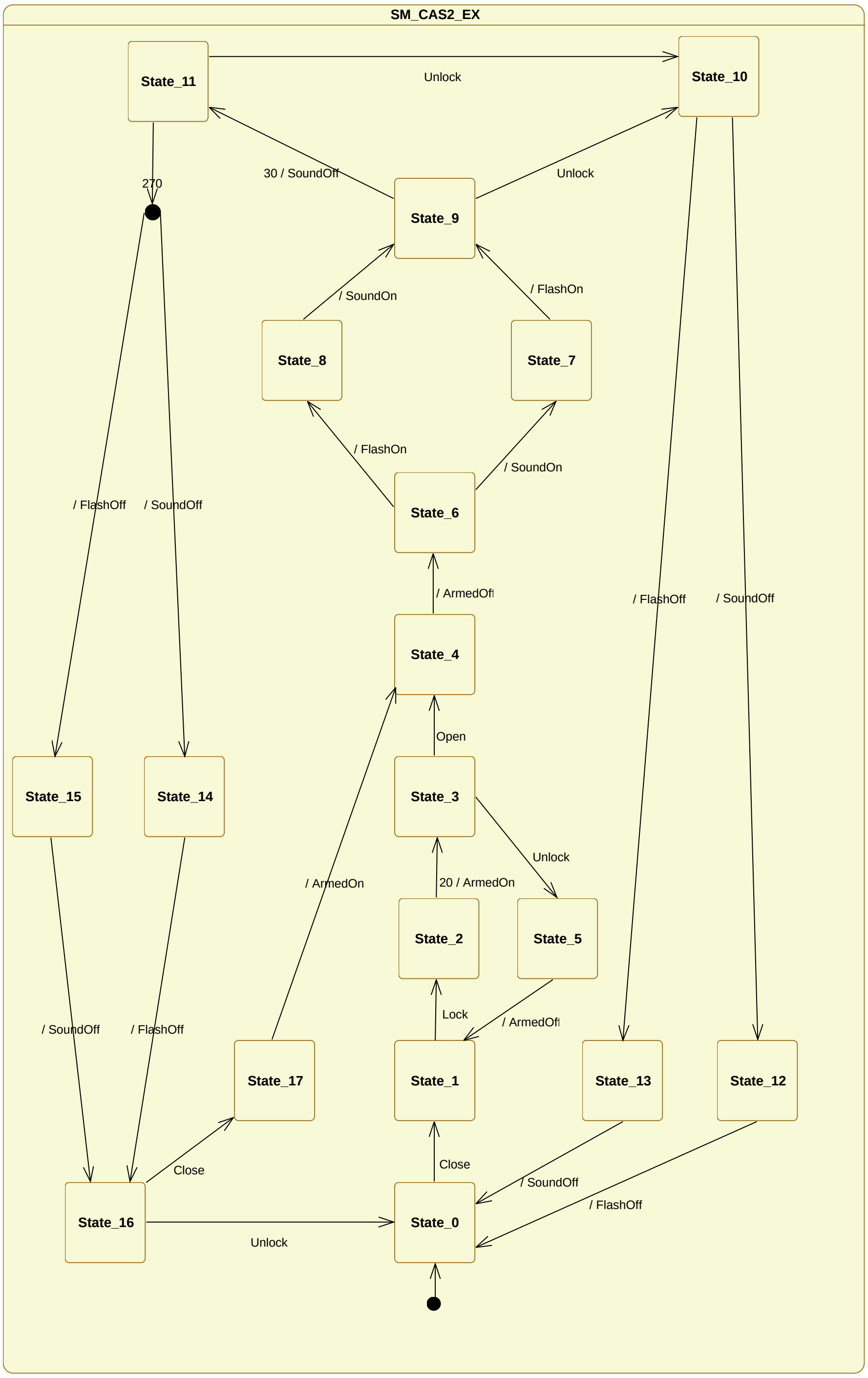


Figure 5.7: State machine, simple state machine version, cas_2

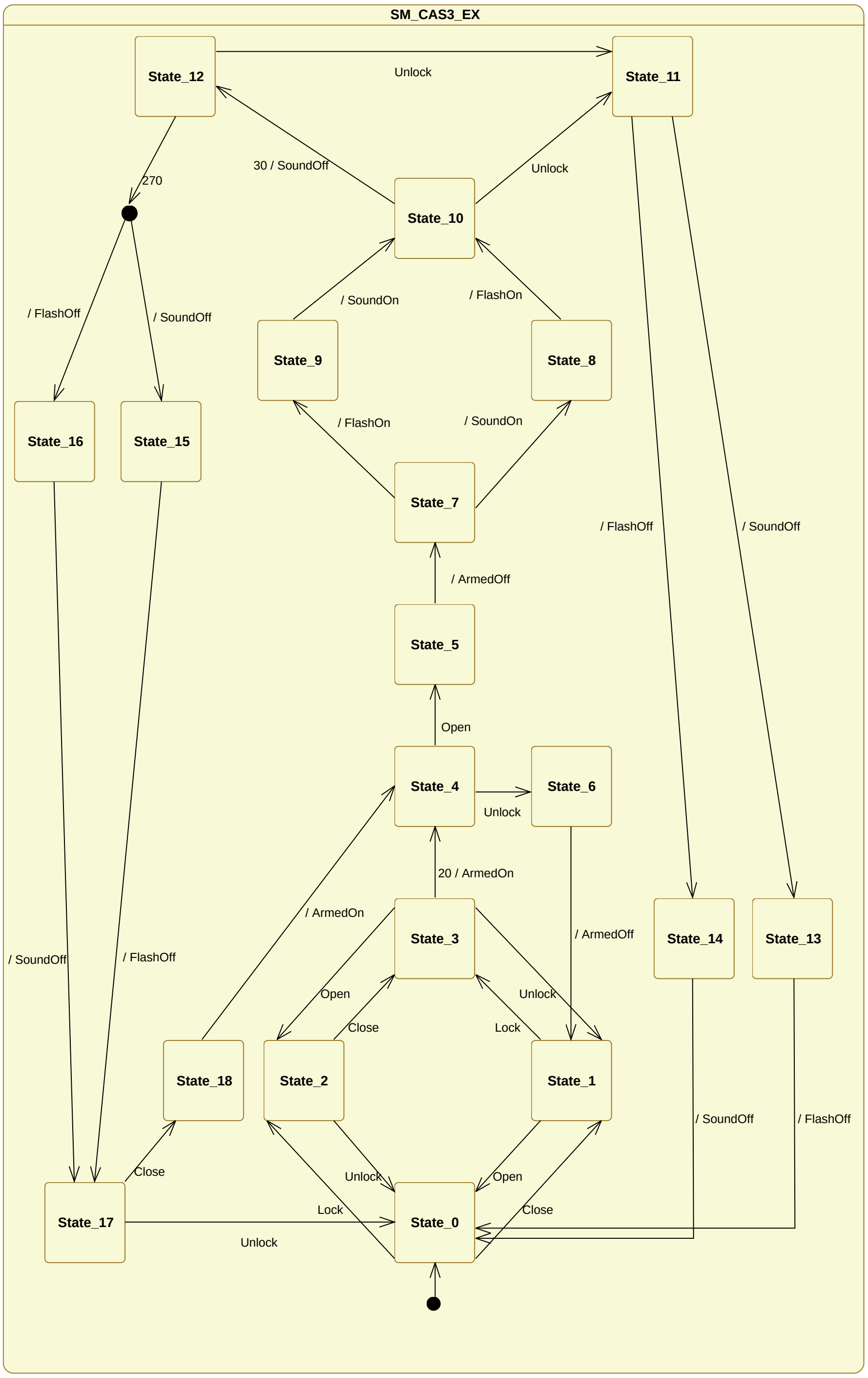


Figure 5.8: State machine, simple state machine version, cas_3

5.2.2 Action System Like Modeling Style

As described in Section 3.1.1 properties of classes can be used as variables. This kind of notion is often referred to as extended state machines.

In extended state machines, whether a transition can be taken, does not only depend on whether the corresponding event triggers it. Instead, also in a so called guard, the evaluation of expressions can enable or disable a transition.

This is the same mechanism which is also used within an object-oriented action system, so one can think of the other way round and translate an OOAS back to a state machine. The models presented in this section are the result of a manual translation of the OOAS models, the author of this thesis has created for the technical report [ALT12b], into UML.

This modeling style resembles more of how one would program a system using an imperative paradigm than how one would model it using a UML state machine diagram.

Figure 5.9 shows the class diagram of the partial model cas_1 and Figure 5.10 shows the class diagrams of the partial model cas_2 which is the same as for the full model cas_3. One important characteristics of this modeling style is the existence of several Boolean flags represented as properties, which encode the state of the system. This includes describing flags like `open`, `locked`, and `armed` as well as flags that encode the next operation to be performed as `armedOn` or `armedOff`. One special variable is the Integer property `semaphore`. In the object-oriented action system this variable had been used to coordinate the input and the output events. Whenever outputs are produced, this variable is set to the number of produced outputs and each time, an observable action is actually chosen this variable is decreased. Only if this variable is zero, an input can occur. When using UML models as input, this synchronization is usually achieved as side-effect of the run-to-completion semantics, which is implemented in *UMMU*. However, omitting the variable and removing its occurrences within the guards and effects would lead to some events in the LTS that would be inconsistent to the Java implementation. This happens due to some specific implementation detail of the UML-to-OOAS transformation regarding the time triggers.

Figure 5.11 shows the first partial model using this extreme kind of modeling style.

Characteristic to this style is, that there is one “big” state with many self-loops. This state represents the **do od**-block, the iterative loop of an action system. Because of the lack of an if-then-else statement, a so-called DNF partitioning [DF93] has been performed. Each self-loop represents a conjunction, which is used as guard.

For instance in the first partial model, there are two cases, when an input event “Close” can occur: it can either occur in a state, where the car is still unlocked or in a state, where the car is already locked. Because of that, there are two self-loops triggered by the Close event: Close1 and Close2. Close1 handles the case that the car is already locked and therefore sets the flag `armedOn`, that enables the 20 seconds timer that triggers the *ArmedOn* output event, while Close2 only sets the `open` flag to false.

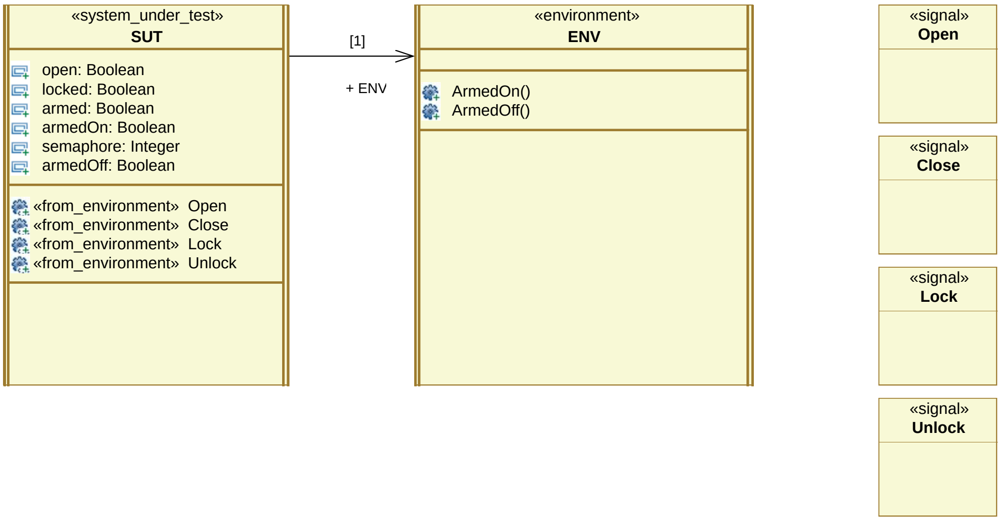


Figure 5.9: Class diagram, action system like version, cas_1

Table 5.1 lists the guards and the opaque behavior of the transition effects, which have been hidden in the state machine diagram for readability reasons.

Figure 5.12 shows cas_2 and Figure 5.13 shows cas_3 using this style.

One interesting effect of this modeling style is, that the corresponding LTS does not conform to either LTS of the former models in terms of *ioco*, even though it is perfectly suitable for generating test cases which can be run against the Java implementations. The reason for this lies again in the UML-to-OOAS transformation of the time events, which required already the workaround using the semaphore. As mentioned in Section 3.2.2 within this transformation only “interesting” time values for the internal `after` are picked in order to keep the generated LTS as simple as possible.

Figure 5.14 shows a comparison of the LTS up to the first input event of both the original model and the model using the “Action System Like” modeling style. The LTS already contain the quiescence labels δ, which are appended by *Ulysses*.

In the state machine of the original model in the initial state there is no outgoing transition triggered by a time event. Therefor no time trigger is registered in the initial state of the generated OOAS, which causes the internal `after` action to be disabled. Hence, the only enabled events in the initial state of the LTS are the input events “Close(0)” and “Lock(0)” with the parameter “0” denoting that no time has passed. Since all outgoing transitions are labeled with an input event, the initial state is a quiescent state.

In the state machine of the model using the “Action System Like” modeling style, from the initial state there are also outgoing transitions fired by time triggers. Even though the transitions are disabled, the corresponding time triggers are still registered in a

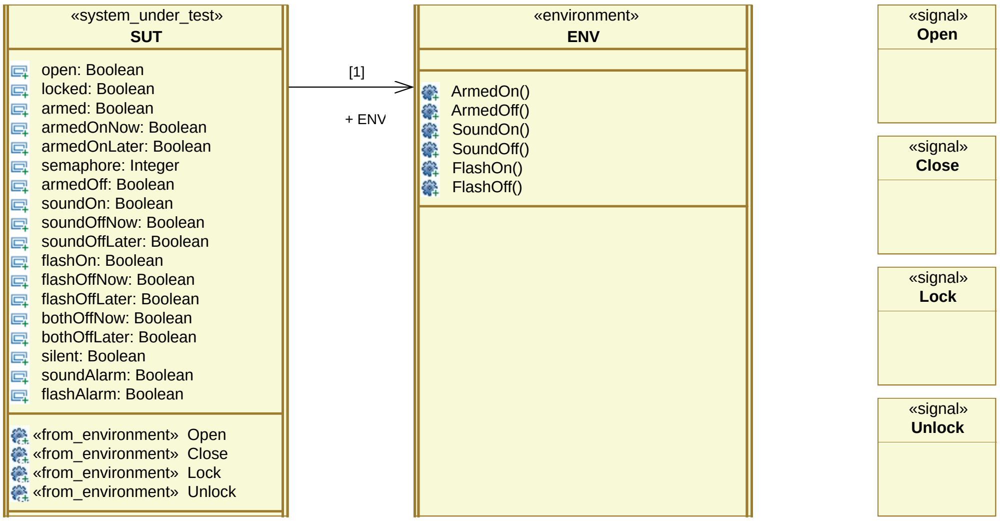


Figure 5.10: Class diagram, action system like version, cas_2 and cas_3

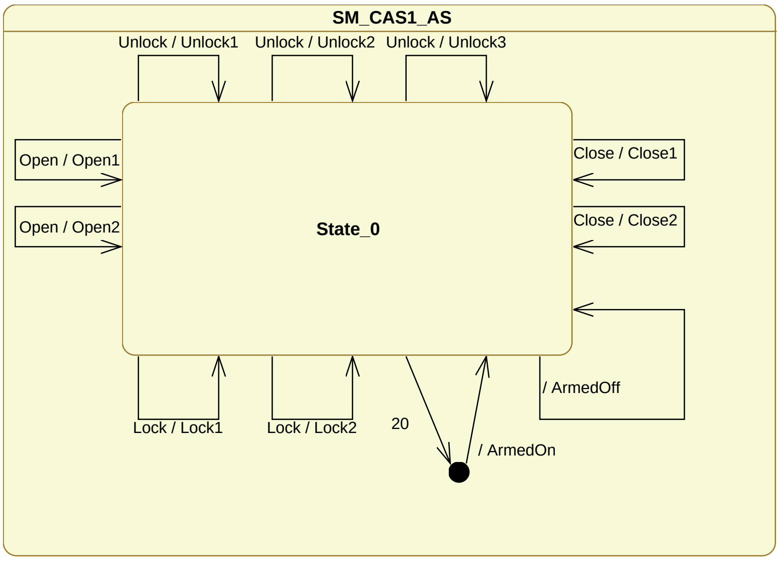


Figure 5.11: State machine, action system like version, cas_1

Trigger	Name	Guard (OCL)	Opaque Behavior (AGSL)

Close	Close1	`armed = false and`
		`open = true and`
		`locked = true and`
		`semaphore = 0`
			`open = false;`
			`armedOn = true;`

Close	Close2	`armed = false and`
		`open = true and`
		`locked = false and`
		`semaphore = 0`
			`open = false;`

none	ArmedOff	`armedOff = true`	`armedOff = false;`
			`semaphore = semaphore - 1;`
			`armed = false;`
			`run ENV.ArmedOff();`

20 time units	Wait20		

none	ArmedOn	`armed = true`	`armedOn = false;`
			`run ENV.ArmedOn();`

Lock	Lock1	`armed = false and`
		`locked = false and`
		`open = true and`
		`semaphore = 0`
			`locked = true;`

Lock	Lock2	`armed = false and`
		`locked = false and`
		`open = false and`
		`sempahore = 0`
			`locked = true;`
			`armedOn = true;`

Open	Open1	`armed = false and`
		`open = false and`
		`armedOn = false and`
		`semaphore = 0`
			`open = true;`

Open	Open2	`armed = false and`
		`open = false and`
		`armedOn = true and`
		`sempahore = 0`
			`open = true;`
			`armedOn = false;`

Unlock	Unlock1	`locked = true and`
		`armed = false and`
		`armedOn = false and`
		`semaphore = 0`
			`locked = false;`

Unlock	Unlock2	`locked = true and`
		`armed = true and`
		`armedOn = false and`
		`semaphore = 0`
			`locked = false;`
			`armedOff = true;`
			`semaphore = 1;`

Unlock	Unlock3	`locked = true and`
		`armed = false and`
		`armedOn = true and`
		`semaphore = 0`
			`locked = false;`
			`armedOn = false;`

Table 5.1: OCL and AGSL code of car alarm system, action system like version, cas_1

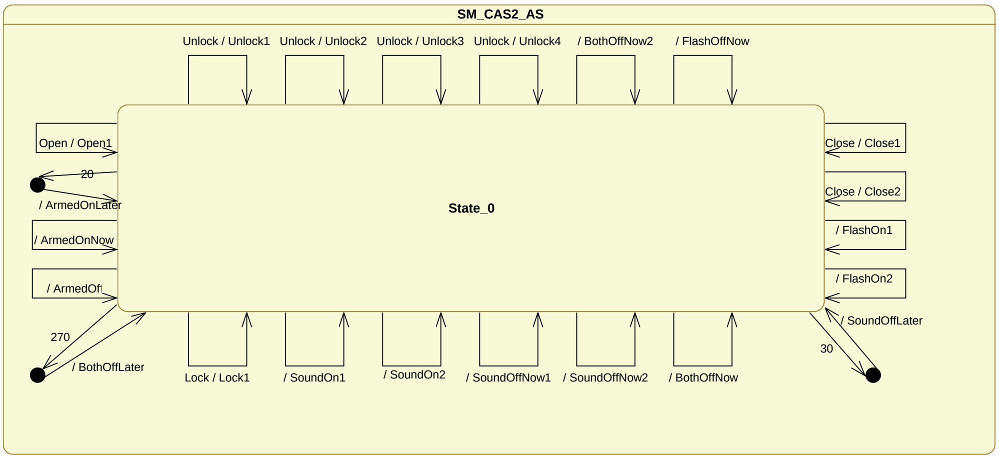


Figure 5.12: State machine, action system like version, cas_2

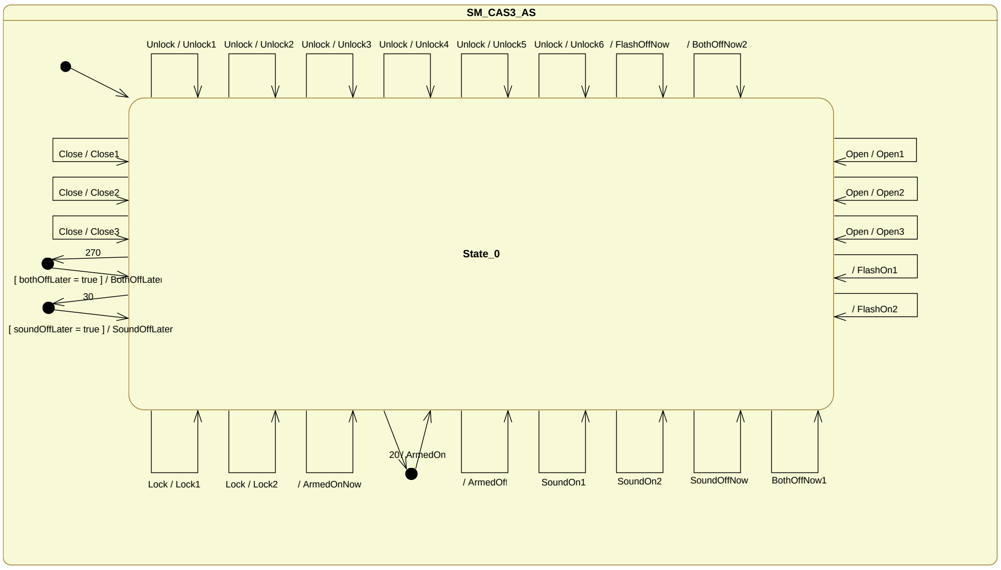


Figure 5.13: State machine, action system like version, cas_3

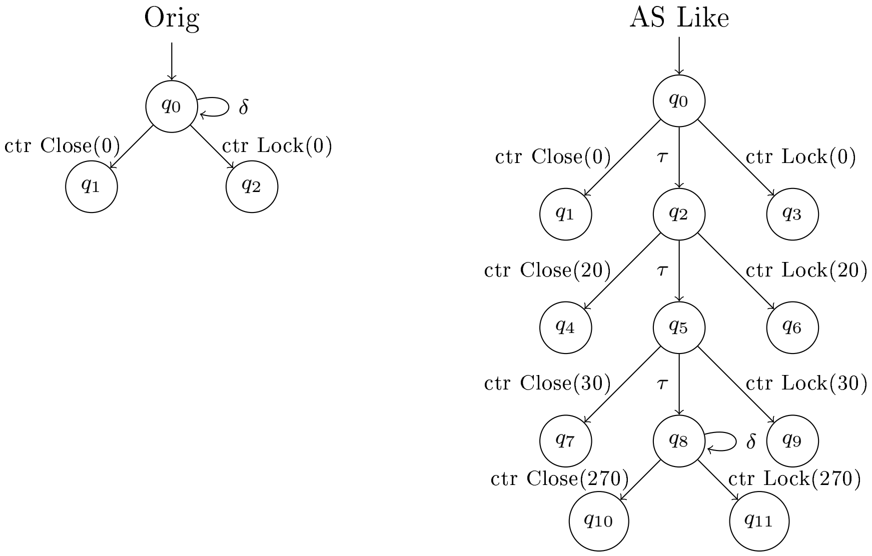


Figure 5.14: Differences in the LTS up to first input event of action system like version compared to other modeling styles

queue. This queue is used by the internal `after` action in order to determine, which time steps are “interesting”. So additionally to the input events “Close(0)” and “Lock(0)” in the **do od**-block of the OOAS, the internal event “after(20)” can be chosen non-deterministically. In the LTS this internal event is represented as τ. Note, that if there is an outgoing transition labeled with τ from a state, *Ulysses* does not consider this state as quiescent, so q_0 is not quiescent.

If the internal event “after(20)” is chosen, the variable storing the lapsed time since the last visible action is increased, leading to state q_2, where the choice is among the events “Close(20)”, “Lock(20)”, and “after(10)”, with “10” being the remaining time to the next registered time trigger. Therefor, also q_2 is not quiescent.

If the internal even is chosen again, this leads to state q_5, where the choice is among “Close(30)”, “Lock(30)”, and “after(240)”, so neither q_5 is quiescent.

The internal event can be chosen one last time leading to state q_8 . There the outgoing transitions are labeled with “Close(270)” and “Lock(270)”, which are both input events, so q_8 finally is a quiescent state.

In order to perform an *ioco* check, the LTS containing the quiescence information has to be determinized, resulting in a suspension automaton. In *Ulysses* this is done by using the standard Rabin-Scott powerset construction algorithm [RS59]. Figure 5.15 shows the suspension automaton up to the first input event. The initial state of the suspension automaton represents all states, that can be reached from the initial state of the underlying LTS by using transitions labeled with τ. Note, that since the self-loop with the quiescence label δ only occurred in one of these states, in the suspension automaton this is represented by a transition to a new state representing only the quiescent state.

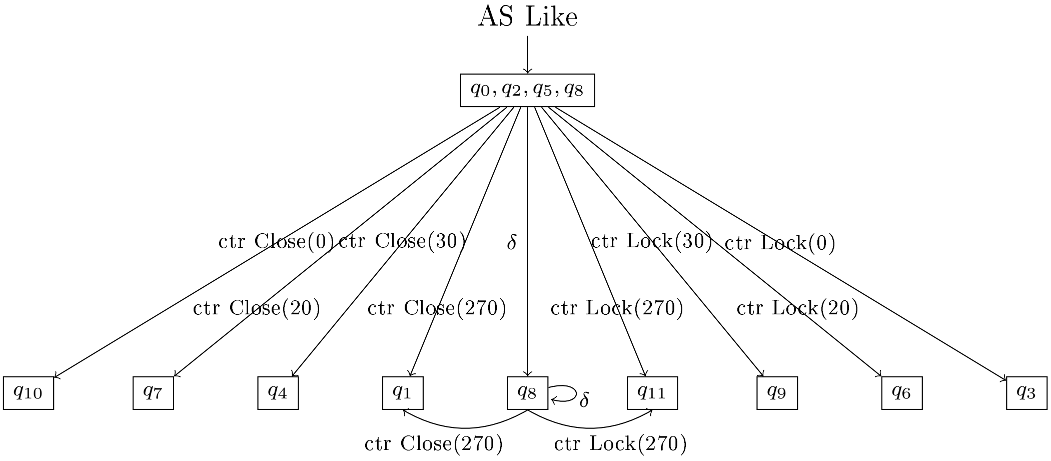


Figure 5.15: Suspension automaton of action system like version up to first input event

As can be seen in the extract of the suspension automaton, the trace $< \delta$, Close(270) $>$ is defined, while the trace $< \delta$, Close(0) $>$ is not. Since the parameter denotes the lapsed time, an intuitive interpretation in the context of software testing would be, that after waiting for a time-out it is not possible to send the "Close" input without delay. For creating test cases, this is not a problem, for conforming to the LTS of the original model it is.

Let us consider the trace $< \delta$, Close(0), Lock(0) $>$, which is a trace of the original model. However, after the first step *delta* the suspension automaton of the model using the "Action System Like" style, will be in a state, where neither "Close(0)" nor "Lock(0)" are defined. Because of its role in conformance checking the automaton is assumed to be input enabled. So the undefined events are ignored and the automaton stays in a quiescent state. The LTS of the original model on the other hand prescribes the output event "ArmedOn(20)" after this trace, so the model using the "Action System Like" modeling style does not conform to the original model in terms of *ioco*.

In summary it can be said, that this is an extreme form, of how a UML state machine can look like. However, it gives a good impression on how action systems, the internal representation within the *MoMuT toolchain*, work.

5.2.3 Multiple Classes Modeling Style

As mentioned in Section 3.1.1 there are two ways of expressing parallel behavior in test models used for the *MoMuT* toolchain: orthogonal regions and multiple classes. Test models using orthogonal regions have been created quite often within the MOGENTES project. For instance, a test model for the wheel loader, which is also investigated as second case study in Chapter 6 of this thesis, uses orthogonal regions. However, for demonstrating the object-oriented features, it makes more sense to use multiple classes. Since it is a test model and not a design model, design principles like data encapsulation have not been considered as important. Instead, the focus lies on reducing the overhead resulting from the UML-to-OOAS transformation. Communication between the objects is done by reading shared variables. Additionally to the variables which are explicitly defined as properties in the class diagram, the current state of another object can be accessed using the Boolean function `oclIsInState(S)`, which returns true, if and only if the object is in the state `S`.

Figures 5.16 resp. 5.17 show the corresponding class diagrams. The partial model cas_1 consists of five classes: “Door”, “Locking System”, “Armed”, “ENV” and “SUT”. The classes “Door” and “Locking System” have the signal receptions for the signals “Close” and “Open” resp. “Lock” and “Unlock”. The class “Armed” controls the outputs “ArmedOn” and “ArmedOff” by calling the corresponding operations on the environment object. The class “ENV” models the environment and provides the operations, which are used to represent the output of the system-under-test. The class “SUT” is just a dummy element with no functionality at all. The *MoMuT* toolchain requires that exactly one class is marked with the stereotype «system_under_test». The alternative would have been to arbitrarily choose another class to mark it with this stereotype.

The partial model cas_2 and the full model cas_3 contain another three additional classes: “FlashAlarm”, “SoundAlarm” and, “Timer”. The classes “FlashAlarm” and “SoundAlarm” control the outputs “FlashOn”, “FlashOff”, “SoundOn”, and “SoundOff” by calling the corresponding operations on the environment object. The class “Timer” is needed to work around an issue within the current version of *UMMU* which requires all time triggers to occur in the state machine of the same class.

Figure 5.18 shows the instance specifications of the full model. Within the UML-to-OOAS transformation they are translated into objects. This is necessary, since in the current implementation of *Argos* objects have to been known at compile time.

Note, that for each object, that accesses the state of another object, there has to be an association in the class diagram. In the diagram representing the instances, these links are represented as initializations of the corresponding “defining features”. In the generated OOAS the “housekeeping object” initializes each object by setting references to each needed other object.

Figures 5.19, 5.20 and 5.21 show the state machines of the partial model in the distributed style. The state machines of the door and the locking system are very simple. They just consist of two states, which are changed, when the corresponding signal is received. The state machine for the “Armed” object watches the state of both door and locking system using a triggerless transition with a guard. As soon as the door is closed and the locking system is locked, the system changes into the “Armed_Closed_And_Locked” state. From there it can either change back to the “Armed_Idle” state if one of the other state machines change their state. Otherwise, after 20 time units the system changes to the “Armed_Armed” state, where as entry action the corresponding output event is produced.

The models for the partial model cas_2 and the full model cas_3 are more complex. As already mentioned, all time triggers have to be in the same class. However, in this test

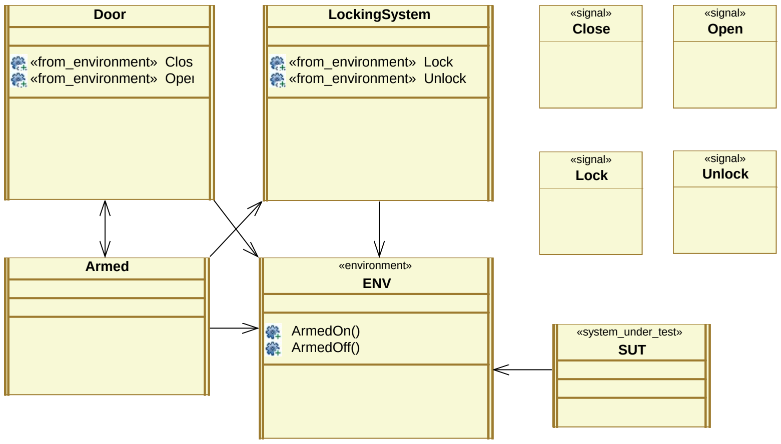


Figure 5.16: Class diagram, multiple classes version, cas_1

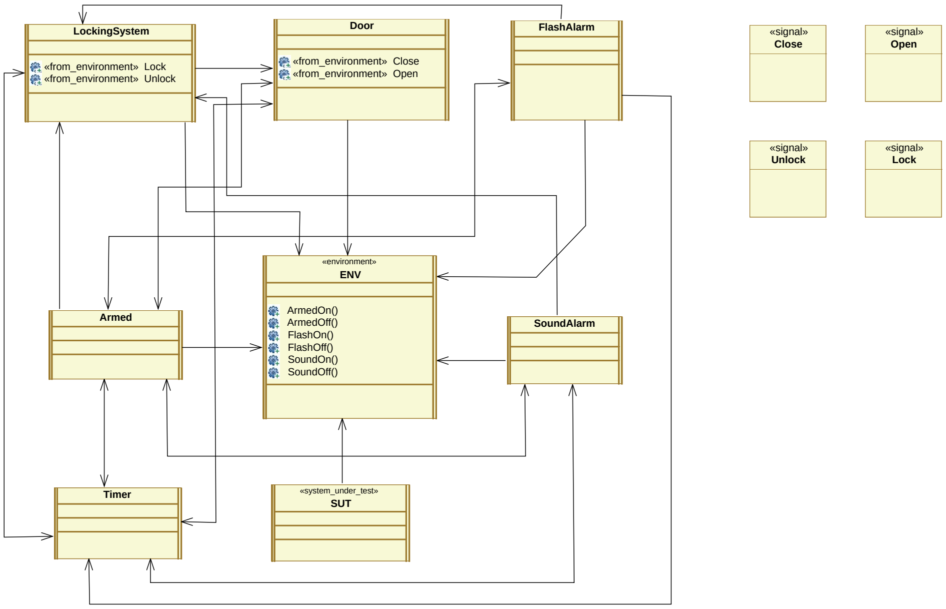


Figure 5.17: Class diagram, multiple classes version, cas_2 and cas_3

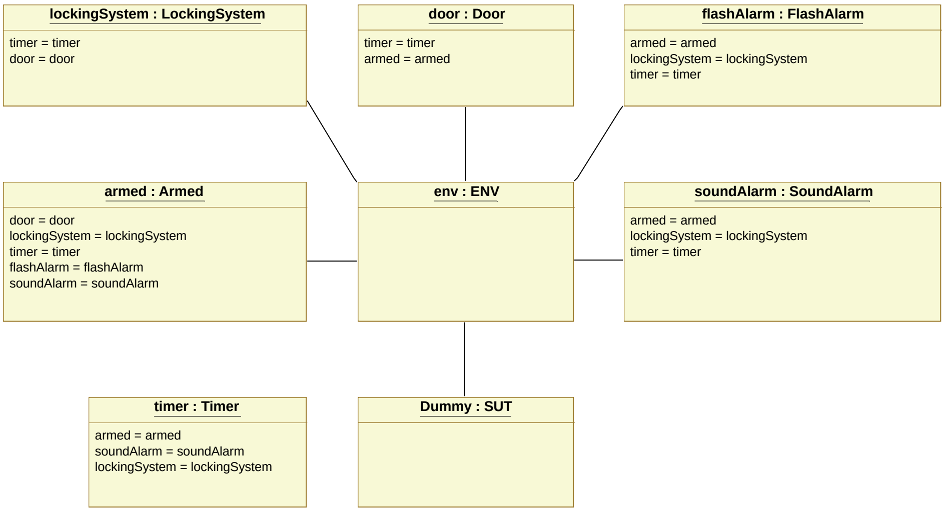


Figure 5.18: Class diagram showing instances, multiple classes version, cas_2 and cas_3

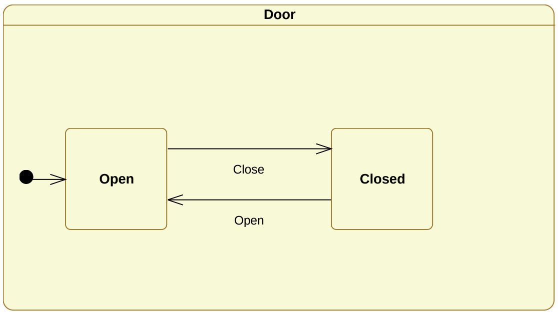


Figure 5.19: State machine door, multiple classes version, cas_1

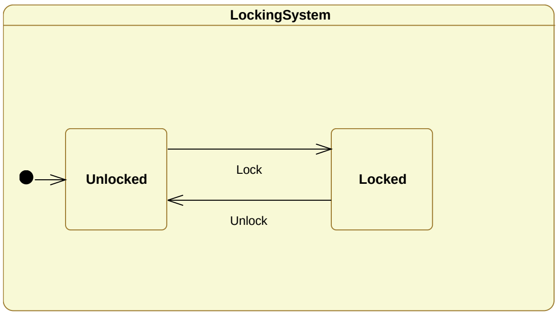


Figure 5.20: State machine locking system, multiple classes version, cas_1 and cas_3

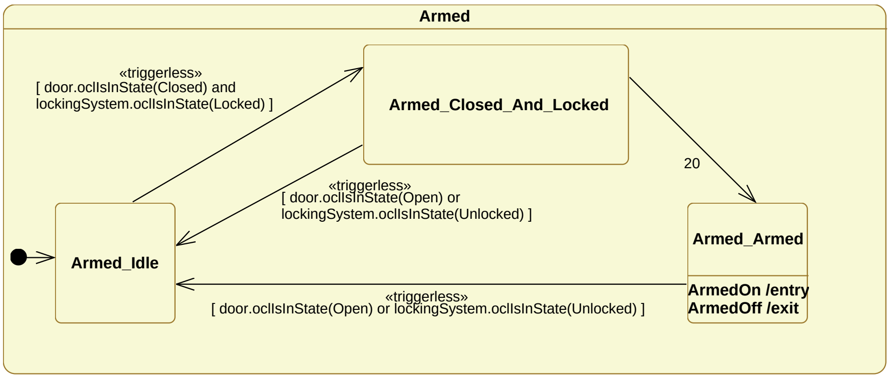

Armed
«triggerless»
[door.oclIsInState(Closed) and
lockingSystem.oclIsInState(Locked)]
Armed_Closed_And_Locked
20
«triggerless»
[door.oclIsInState(Open) or
lockingSystem.oclIsInState(Unlocked)]
Armed_Armed
Armed_Idle
ArmedOn /entry
ArmedOff /exit
«triggerless»
[door.oclIsInState(Open) or lockingSystem.oclIsInState(Unlocked)]

Figure 5.21: State machine armed, multiple classes version, cas_1

model there are three classes, in which the lapse of time is modelled. The object of the “Armed” class (depicted in Figure 5.22) starts a timer of 20 time units when it is in the “Armed_Closed_And_Locked” state before going to the “Armed_Armed” state, the object of the “SoundAlarm” class (depicted in Figure 5.24) has to wait 30 time units in the “SA_Sound_And_Flash_Alarm” state before turning of the sound alarm and switching to the “SA_Flash_Alarm” state, while the “FlashAlarm” class (depicted in Figure 5.25) has to wait for 270 time units before turning off the Flash Alarm. The solution is to create an additional “Timer” class (depicted in Figure 5.23), which implements all timers and use triggerless transitions with guards checking the state of the other classes in order to synchronize the state machines.

This test model might not look very nice because of the workarounds needed for the time handling and the redundancy in the state machines, but it has one notable characteristic: the non-determinism in terms of the two possible interleavings when turning on or off the Alarms, is expressed implicitly using objects, which run in parallel.

Another interesting observation is that unlike in every other modeling style, the partial model cas_2 is not smaller than the model cas_3, but actually more complex. This is due to the fact, that defining only one path to the point, where the “Armed” output event is sent in the underlying LTS, is done by explicitly disabling the occurrence of the other possible input events in the guard of the corresponding transitions in the UML state machines. While model cas_3 uses the same state machines for the locking system as the first partial model, in the second partial model there are additional guards as depicted in Figure 5.27.

Note, that even in the model cas_3 there are states, in which the input event “Close” is not specified, even though the door is in a “Open” state. Figure 5.26 shows the state machine of the door of model cas_2 and Figure 5.28 shows the corresponding state machine of model cas_3.

Ulysses can be used to proof the conformance to the test models using other styles. The LTS generated from this model conforms to the LTS of the simple state machine,

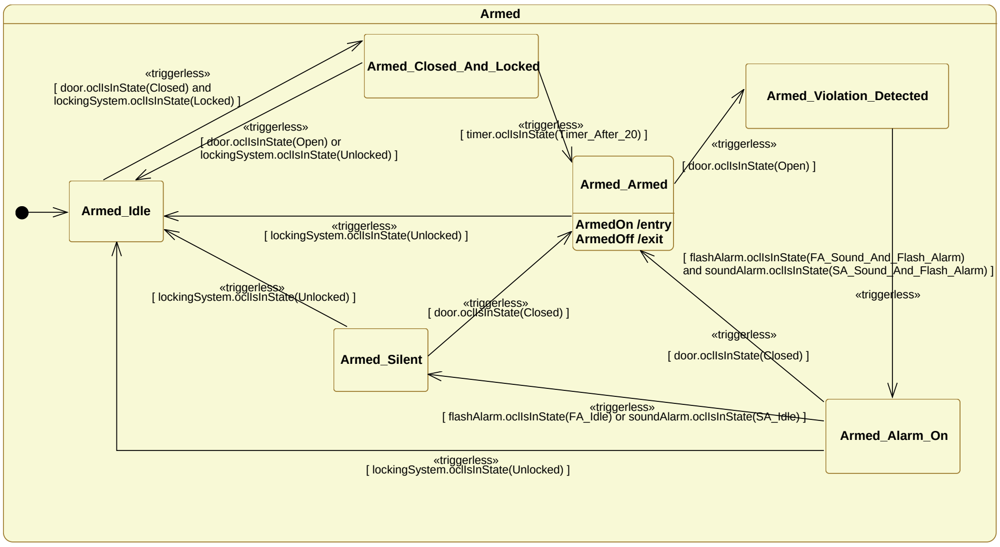

Armed
Armed_Closed_And_Locked
«triggerless»
[door.oclIsInState(Closed) and
lockingSystem.oclIsInState(Locked)]
«triggerless»
[door.oclIsInState(Open) or
lockingSystem.oclIsInState(Unlocked)]
«triggerless»
[timer.oclIsInState(Timer_After_20)]
Armed_Violation_Detected
«triggerless»
[door.oclIsInState(Open)]
Armed_Idle
Armed_Armed
ArmedOn /entry
ArmedOff /exit
«triggerless»
[lockingSystem.oclIsInState(Unlocked)]
[flashAlarm.oclIsInState(FA_Sound_And_Flash_Alarm)
and soundAlarm.oclIsInState(SA_Sound_And_Flash_Alarm)]
«triggerless»
«triggerless»
[lockingSystem.oclIsInState(Unlocked)]
«triggerless»
[door.oclIsInState(Closed)]
«triggerless»
[door.oclIsInState(Closed)]
Armed_Silent
«triggerless»
[flashAlarm.oclIsInState(FA_Idle) or soundAlarm.oclIsInState(SA_Idle)]
Armed_Alarm_On
«triggerless»
[lockingSystem.oclIsInState(Unlocked)]

Figure 5.22: State machine armed, multiple classes version, cas_2 and cas_3

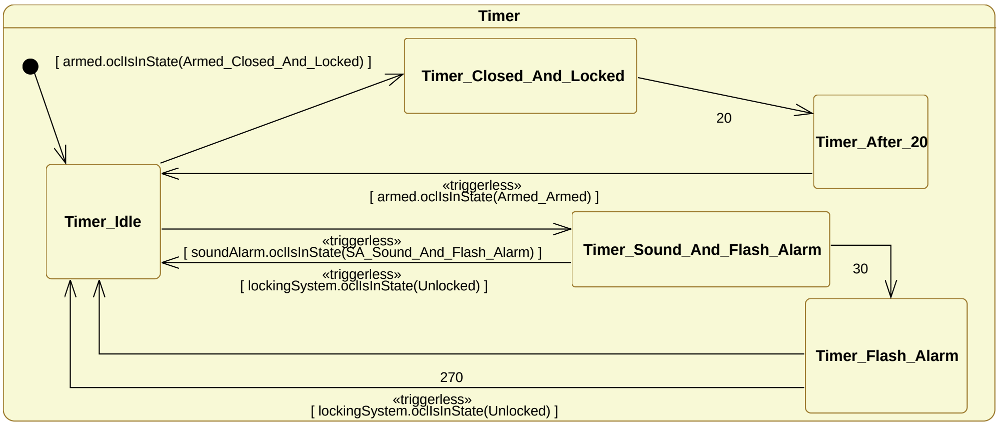

Timer
[armed.oclIsInState(Armed_Closed_And_Locked)]
Timer_Closed_And_Locked
20
Timer_After_20
Timer_Idle
«triggerless»
[armed.oclIsInState(Armed_Armed)]
«triggerless»
[soundAlarm.oclIsInState(SA_Sound_And_Flash_Alarm)]
Timer_Sound_And_Flash_Alarm
«triggerless»
[lockingSystem.oclIsInState(Unlocked)]
30
Timer_Flash_Alarm
270
«triggerless»
[lockingSystem.oclIsInState(Unlocked)]

Figure 5.23: State machine timer, multiple classes version, cas_2 and cas_3

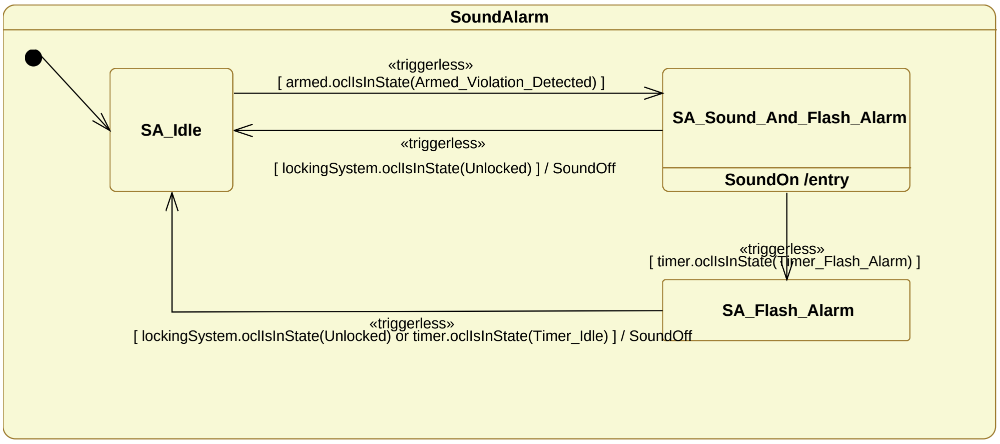

SoundAlarm
SA_Idle
«triggerless»
[armed.oclIsInState(Armed_Violation_Detected)]
SA_Sound_And_Flash_Alarm
«triggerless»
[lockingSystem.oclIsInState(Unlocked)] / SoundOff
SoundOn /entry
«triggerless»
[timer.oclIsInState(Timer_Flash_Alarm)]
SA_Flash_Alarm
«triggerless»
[lockingSystem.oclIsInState(Unlocked) or timer.oclIsInState(Timer_Idle)] / SoundOff

Figure 5.24: State machine sound alarm, multiple classes version, cas_2 and cas_3

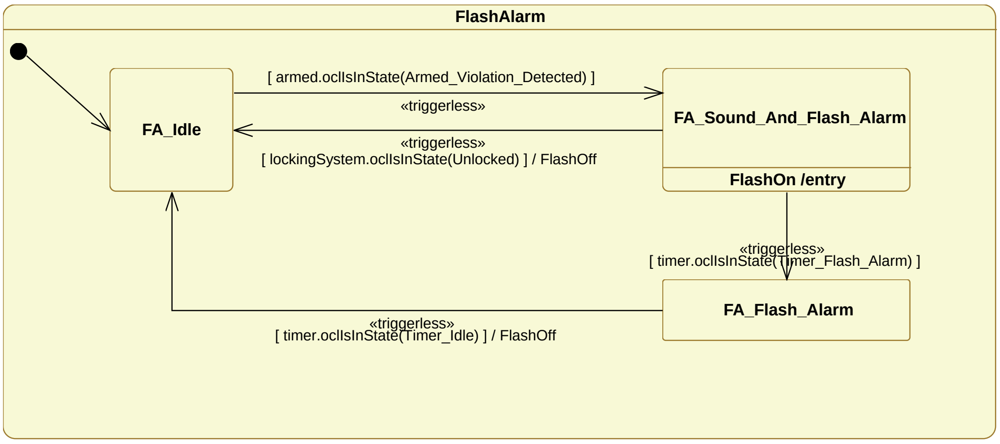

FlashAlarm
FA_Idle
[armed.oclIsInState(Armed_Violation_Detected)]
«triggerless»
FA_Sound_And_Flash_Alarm
«triggerless»
[lockingSystem.oclIsInState(Unlocked)] / FlashOff
FlashOn /entry
«triggerless»
[timer.oclIsInState(Timer_Flash_Alarm)]
FA_Flash_Alarm
«triggerless»
[timer.oclIsInState(Timer_Idle)] / FlashOff

Figure 5.25: State machine flash alarm, multiple classes version, cas_2 and cas_3

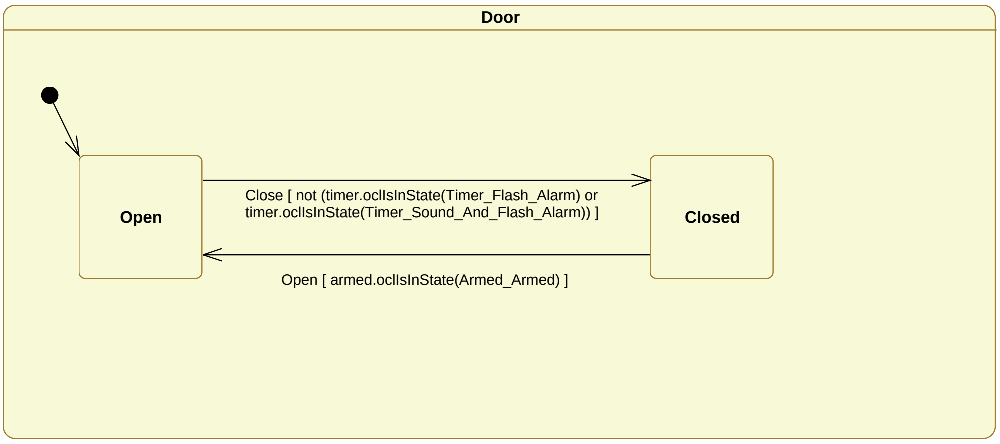

Door
Open
Closed
Close [not (timer.oclIsInState(Timer_Flash_Alarm) or timer.oclIsInState(Timer_Sound_And_Flash_Alarm))]
Open [armed.oclIsInState(Armed_Armed)]

Figure 5.26: State machine door, multiple classes version, cas_2

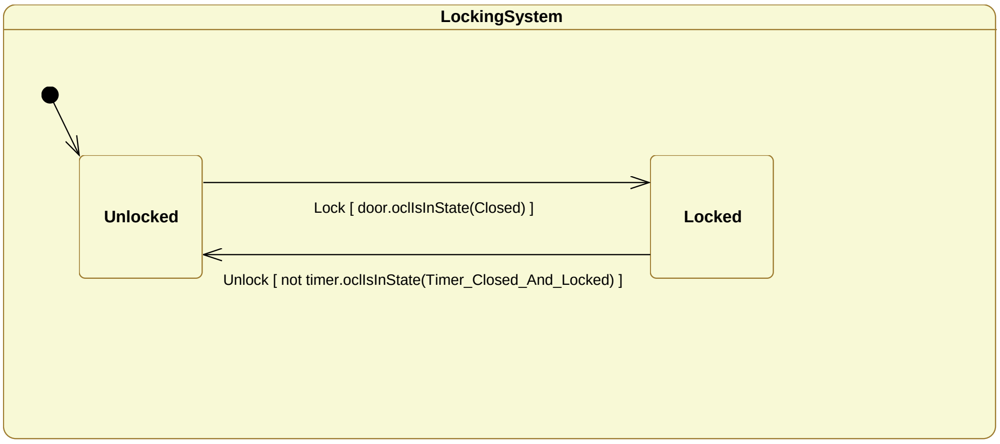

LockingSystem
Unlocked
Locked
Lock [door.oclIsInState(Closed)]
Unlock [not timer.oclIsInState(Timer_Closed_And_Locked)]

Figure 5.27: State machine locking system, multiple classes version, cas_2

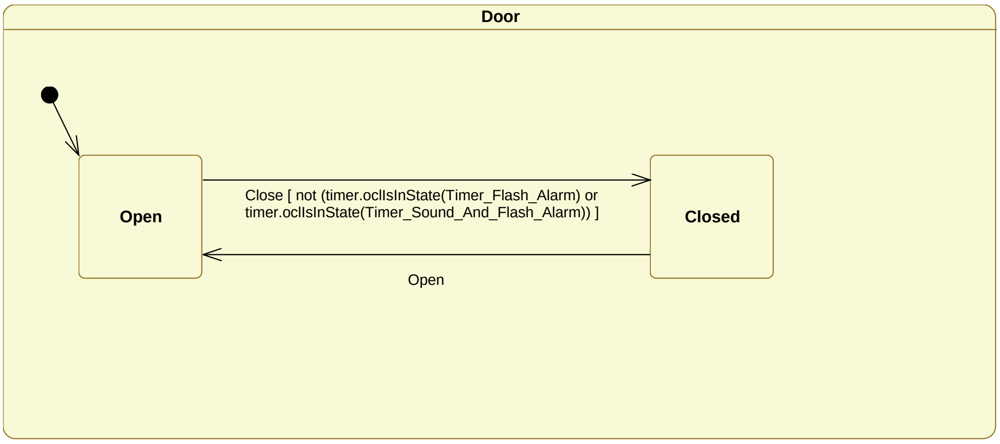

Door
Open
Closed
Close [not (timer.oclIsInState(Timer_Flash_Alarm) or timer.oclIsInState(Timer_Sound_And_Flash_Alarm))]
Open

Figure 5.28: State machine door, multiple classes version, cas_3

described in Section 5.2.1 and vice versa. The LTS of the deterministic model conforms to the LTS of this model, while the LTS of this model does not conform to the deterministic model for the very same reason as with the simple state machine.

5.3 Empirical Results

5.3.1 Application of Mutation Operators

The number of mutants, generated by a fixed set of mutation operators, can be seen as fault-based metrics to measure the complexity of a test model. Each application of a mutation operator represents the anticipation of a possible design flaw. The more often a mutation operator can be applied, the more flaws can be anticipated. Therefor it can be argued, that a test model, which yields many different mutants is more complex than a test model yielding less different mutants.

Table 5.2 shows how many mutants have been generated for each model. For each modeling style there is column, beginning with “Original”, which refers to the style of the original model, presented in Section 1.6. Further columns refer to the three new presented modeling styles “Simple State Machine”, “Action System Like” and “Multiple Classes”. There is each a row for both partial models cas_1 and cas_2 as well as one for the full model cas_3.

Table 5.3 shows in detail how often each mutation operator has been applied to each full model cas_3.

Model	Original	Simple SM	AS Like	Multiple Classes
cas_1	58	59	419	44
cas_2	70	369	1185	174
cas_3	100	399	1648	168

Table 5.2: Generated mutants per model of the car alarm system.

When comparing the number of generated mutants the original model created by AIT is the smallest model. Most effects are modeled as entry and exit actions, only one is linked to a transition. Because of that, in this model the mutation operators on removing and mutating entry and exit actions are applied. However, these mutation operators keep these effects atomically bound to the state. There is no mutation operator that enables or disables such an action depending on which transition leads towards or away from the respective state. The one effect which is linked to a transition leads to exactly one mutation, in which this effect is removed. Since there are no pairs of transitions having effects, the mutation operator for pairwise swapping the effect can not be applied. This is totally different for the “Simple State Machine” model, described in Section 5.2.1, where all states and transitions are drawn explicitly. Here, all the effects are linked to the transitions and none is linked to a state, leading to 17 mutants when removing each effect once and 272 transitions when swapping the effects of the transitions pairwise. Even more extreme is the “Action System like” modeling style. Since here the state of the system is represented by attributes, also setting the new state is represented by effects. This results in a total of 702 mutants just using this one mutation operator. Despite of having some redundancy in the “Multiple Classes” modeling style, the number

Mutation Operator	Original	Simple SM	AS Like	Multiple Classes
Removing Trigger Events on all Transitions	15	17	17	7
Mutating Transition Signal Events	36	42	42	4
Mutating Transition Time Trigger Events	18	18	18	18
Mutating Transition OCL Expressions	0	0	54	0
Mutating Transition AGSL Expressions	0	0	48	0
Mutating Guards on All Transitions	19	33	89	93
Mutating OCL Change Expression on all Trans.	0	0	0	0
Removing Entry and Exit Action in all States	4	0	0	5
Mutating Entry and Exit Action in all States	4	0	0	12
Removing Effect in all Transitions	1	17	27	4
Mutating (Replacing) Effect in all Transitions	0	272	702	12
Mutating sub expressions in OCL expressions	0	0	351	3
Mutating Operator in OCL Expressions	0	0	131	6
Mutating AGSL Expressions	3	0	90	4
Total	100	399	1648	168

Table 5.3: Application of mutation operators on cas_3 of the car alarm system.

of generated model mutants is quite moderate with 168 mutants in total. Most of them are generated by mutating the guards on all transitions, as many guards are used in this modeling style in order to synchronize between the state machines.

5.3.2 Results Using Default Strategy

Test Case Generation The principal of generating test cases is based on the development process described in Section 4.3. That means that for each modeling style first the two partial models are investigated and then the full model is used to supplement the obtained test suite.

As described in Section 3.4.2 a variety of different test case selection strategies exist in *Ulysses*. In the technical report [ALT12b] only the strategy S5 ("Lazy Ignorant Killing Strategy") has been investigated and stopping the exploration at the depth of the first unsafe state has been turned on. In this strategy for each mutant *Ulysses* first checks, whether one of the already existing test cases is able to "kill" it by revealing a fault when executing a test case. This strategy supports the attempt to benefit most from refinement by only generating test cases covering aspects which have been introduced in a higher refinement level. Because of that, this strategy can be considered the default strategy in the $MoMuT$ toolchain.

Table 5.4 shows the number of generated test cases per refinement level and per model version. The row Δcas_3 refers to the set of additionally created test cases for cas_3 starting with a test suite, which contains already the test cases from the two partial models cas_1 and cas_2. These test cases are created because there are some mutants of the full model, which cannot be killed by any existing test case. The row $cas_1 \cup cas_2 \cup \Delta cas_3$ is the sum of the three rows above and represents the test suite that results from the regression based approach. For comparison, there is also the row cas_3 which represents the test suite, that results from generating test cases from the full model without using test cases of the partial models.

A rather surprising observation from this data is that starting with the test cases of the partial models when processing the full model (table row Δcas_3) in two out of four modeling styles did not add any additional test case.

Test Suite	Original	Simple SM	AS Like	Multiple Classes
cas_1	9	10	41	6
cas_2	6	11	61	7
Δcas_3	0	0	12	4
$cas_1 \cup cas_2 \cup \Delta cas_3$	15	21	114	17
cas_3	13	20	70	11

Table 5.4: Size of generated test suites, car alarm system, default strategy.

Test Case Execution To evaluate the quality of the generated test suites, the test cases have been executed on the Java implementation and the faulty implementations. These implementations have been reused from the experiments of [Aic+11a]. Note, that they have been already manually filtered so that only 38 unique faulty implementations are left.

Test Driver and Test Adapter Most parts of the test driver and test adapter could be reused from the former experiments. This test driver uses calls of methods to communicate with the implementation under test. Thus it implements the Java interface of the environment, which includes Java methods corresponding to the output Operations *ArmedOn*, *ArmedOff*, *FlashOn*, *FlashOff*, *SoundOn*, and *SoundOff*.

Abstract test cases are parsed and executed. Controllable actions are processed by calling the corresponding methods, responses of the implementation are received by the call-back methods and stored by the test driver. When parsing an observable action, it is compared to the stored event from the test driver in terms of *ioco*. If the observation received by the call-back method is allowed by the test case, the test case is continued, otherwise the test case kills the implementation and a “fail” verdict is given.

The lapse of time is simulated by a discrete *Tick()* method in the implementation. In the test case it is presented in the first parameter of each action. For controllable actions the test driver calls the *Tick* method repeatedly according to the time denoted by the parameter. For observable actions, the *Tick* method is called until an observable is received by the call-back functions or a time out has occurred, which is interpreted as quiescence. The information is then stored in the expected label, which is used to determine the conformance.

New with this experiments, using now also models of non-deterministic systems, is the existence of cyclic test cases. These are created when a mutation leads to an unsafe state only in a branch, which the system is allowed to enter non-deterministically. Figure 5.29 depicts a schematic LTS of a specification “O” and a mutant “M” which yield such a cyclic test case “TC”.

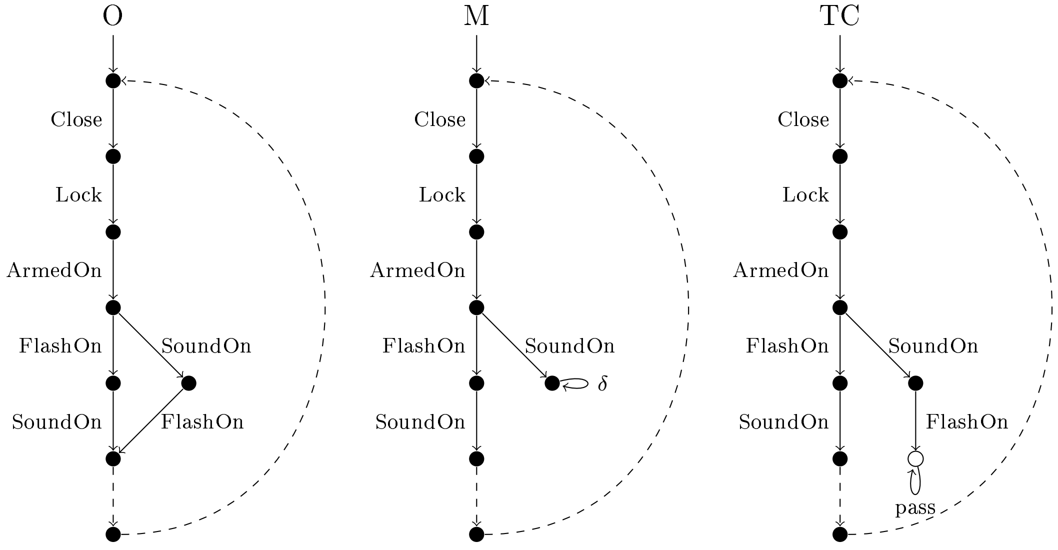


Figure 5.29: A cyclic test case

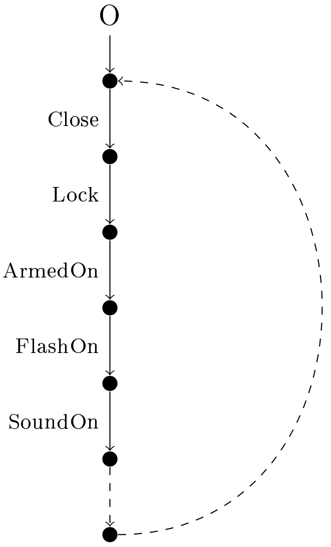


Figure 5.30: Deterministic implementation

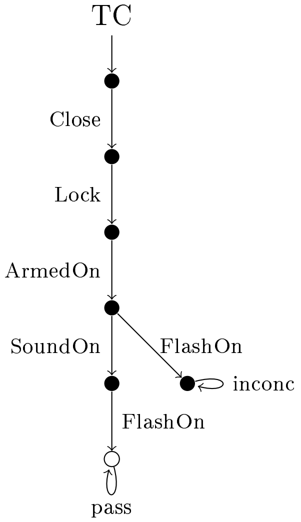


Figure 5.31: A linear test case

Test Suite	Original	Simple SM	AS Like	Multiple Classes
cas_1	71 %	71 %	82 %	66 %
cas_2	63 %	66 %	74 %	66 %
$cas_1 \cup cas_2$	84 %	87 %	100 %	82 %
$cas_1 \cup cas_2 \cup \Delta cas_3$	84 %	87 %	100 %	87 %
cas_3	84 %	87 %	100 %	87 %

Table 5.5: Mutation scores of the generated test suites using the default strategy on 38 faulty Java implementations.

If this test case is executed on a deterministic implementation as depicted in Figure 5.30, there is an infinite loop, since the implementation will never enter the branch, in which the pass state occurs. This is a consequence of using non-determinism only in terms of under-specification, since there is no fairness-assumption, which would exclude this case.

Because of that, a time out or a loop counter has to be implemented in the test driver. If the loop is traversed too often, an inconclusive verdict is given. Note, that a linear test case, as depicted in Figure 5.31 would have an explicit inconclusive verdict.

Discussion The result of a mutation analysis is the so-called "mutation score". The mutation score is defined as ratio of the killed faulty implementations over the total number of non-equivalent mutants. As already mentioned, the equivalent mutants had already been removed from the set of faulty implementations, which are reused in this experiment.

The most powerful test suite is the one generated from the model following the "Action System Like" modeling style. It is the only one able to kill all 38 faulty implementations. Furthermore, it is not even necessary to generate test cases from the full model. The union of the test suites generated from both partial model is already sufficient.

There are many reasons, why the "Action System Like" modeling style is powerful for the aim of killing mutated implementations. First of all, this model is quite complex, allowing many possible mutations. Other than the models, which explicitly enumerate the possible states, here the actual state is composed of many variables. Mutating the update statements leads to complete new states, that might lead to quite subtle failures. Second, a lot of mutation operators have been implemented for guards in OCL and variable updates in AGSL. This is because they are well known from existing mutation tools. Using a style, where a lot of behavior is represented by these elements, leads to a big test suite. Last but not least, the structure of the "Action System Like" model resembles a lot the way code is written using an imperative programming paradigm. Since the implementation that has been used to run the test suite against is written in Java, one could argue, that here just two different suites of mutation operators have been compared.

On the other hand, one can ask, why the test cases of the other models miss some of the faulty implementations. One explanation for this would be the lack of some specific powerful mutation operators. Other work of Aichernig et al. [ALN13] for instance suggests, that a mutation operator creating sink states is very powerful. Such an operator is still missing in the *MoMuT* toolchain.

Another explanation for missing faulty implementations could be, that the test selection strategy is too weak. This hypothesis can be supported by comparing the results to the results of the publication presenting the different strategies [Aic+11a]. There, using the strategy S3 and S4 resulted in a test suite, which was able to kill all faulty implementations. Even the strategy S5 resulted in a test suite, which missed only one out of 38 faulty implementations. Even though the numbers cannot directly be compared, as the notion of time has changed too, this still indicates, that the decision, to stop the generation of the product graph at the depth of the first unsafe state, has a negative effect on the power of the generated test suites.

5.3.3 Comparison with other Killing Strategies

As described in Section 3.4.2 *Ulysses* can be configured to use other killing strategies than the default one. Note, that the effects of these different strategies had already been investigated and published [Aic+11a]. However, this publication had been based on a previous version of *Ulysses*. Back then, *Ulysses* supported the generation of linear test cases, which was discontinued very soon, because of the huge number of generated test cases and the limited exploration depth. On the other hand, the option to cut the search tree on the depth of the first unsafe state was not yet introduced.

Test Case Generation For the context of this thesis it is interesting, whether some killing strategies of *Ulysses* are able to create more powerful test suites. As result of Section 5.3.2 only the test models following the "Action System Like" modeling style, were able to generate a test suite killing all available faulty implementations.

In order to investigate, whether other test case extraction strategies are able to achieve better results for the modeling styles "Original", "Simple State Machine", and "Multiple Classes", the test case generation process has been repeated with them. For the "Action System Like" modeling style this has also been tried, but cancelled because it did not scale at all. For example, using the strategy S3 without the cut of the search tree, it was not possible to process the first ten out of 1658 mutants within three days. Among the first seven mutants there is one, which takes alone 38 hours to be processed, which has to be considered as too slow for such a simple system.

Killing Strategy	Search Tree Cut	Original	Simple SM	Multiple Classes
S3	yes	126	388	215
S3	no	469	585	700
S4	yes	19	28	28
S4	no	130	100	135
S5	yes	13	20	11
S5	no	33	21	36

Table 5.6: Number of generated test cases per killing strategy (cas_3)

Killing Strategy	Search Tree Cut	Original	Simple SM	Multiple Classes
S3	yes	3.6	16.3	19.1
S3	no	9.0	22.8	25.3
S4	yes	3.4	16.6	19.4
S4	no	31.5	27.7	37.6
S5	yes	3.5	16.3	15.7
S5	no	9.8	34.8	37.3

Table 5.7: Total time t in minutes for generating test suites per different killing strategy (cas_3)

Note, that the killing strategy S3 is not designed for a regression based approach, since it does not depend on existing test cases, whether new test cases are generated. So first a comparison among the strategies S3 – S5 has been made on the full model cas_3.

Table 5.6 shows the size of each generated test suites.

Table 5.7 shows the according run-times for generating the test suites of the full models per killing strategy.

Data presented in these tables suggests, that enabling the cut of the search tree has a big influence on both size of the generated test suites and time needed for the generation process. The biggest differences in terms of the run-time can be found in strategy S4 which is quite as fast as S3 when the cut of the search tree is enabled, while it as significantly slower, when it is disabled.

Additionally, both variants of strategies S4 and S5 can be used to generate test cases via the regression based approach beginning with the partial models. First test suites for cas_1 and cas_2 are generated independently, then they are merged and used as initial test suite for generating test cases for the model cas_3. Table 5.8 shows the size of the generated test suits for this approach.

Test Case Execution Table 5.9 relates the mutation score of each generated test suits to its size. The strategies S3 and S4 without stopping the search at the depth of the

Killing Strategy	Search Tree Cut	Original	Simple SM	Multiple Classes
S4	yes	22	30	34
S4	no	136	105	143
S5	yes	15	21	17
S5	no	27	24	28

Table 5.8: Number of generated test cases per killing strategy using the regression based approach $cas_1 \cup cas_2 \cup \Delta cas_3$.

Killing Strategy	Search Tree Cut	Original		Simple SM		Multiple Classes	
		Size	MS	Size	MS	Size	MS
S3	yes	126	84 %	388	87 %	215	92 %
S3	no	469	100 %	585	100 %	700	100 %
S4	yes	19	84 %	28	87 %	28	92 %
S4	no	130	100 %	100	100 %	135	100 %
S5	yes	13	84 %	20	87 %	11	87 %
S5	no	33	98 %	21	92 %	36	98 %

Table 5.9: Number of generated test cases and mutation scores per killing strategy applied on cas_3

first fail states achieve 100 % mutation score. Since in strategy S4, the generation of test cases for unsafe states, which are already covered by existing test cases is avoided, the size of the test suites decreases significantly. In strategy S5, where the generation of new test cases is already omitted, if any non-conformance between the original model and the model mutant can be shown with existing test cases, the size of the test suites decreases again, but they do not achieve a perfect mutation score anymore. The cut of the search tree decreases the size of each test suite, but the mutation score suffers even more.

Table 5.10 shows the resp. numbers for the regression based approach, where first test cases are generated using the partial models. The mutation scores in most cases are the same, only one number differs slightly for strategy S5, which is very sensitive about the exact order, in which the model mutants are processed. Also in the size of the test suites there is no big difference. This indicates, that if strategy S4 or S5 is used, there is no objection to use the regression based approach.

Discussion The comparison between the different test case extraction methods shows big differences in run-time of the generation process as well as size and quality of the generated test suites. It suggests, that the right choice of the parameters has a bigger influence than the used modeling style. Using the killing strategy S3 or S4 without the cut of the search tree ensures that for each unsafe state there is at least one test case covering it. These test suites are also able to kill all 38 faulty implementations. As

Killing Strategy	Search Tree Cut	Original		Simple SM		Multiple Classes	
		Size	MS	Size	MS	Size	MS
S4	yes	22	84 %	30	87 %	34	92 %
S4	no	136	100 %	105	100 %	143	100 %
S5	yes	15	84 %	21	87 %	17	87 %
S5	no	27	97 %	24	92 %	28	95 %

Table 5.10: Mutation scores per killing strategy using a regression based approach $cas_1 \cup cas_2 \cup \Delta cas_3$.

downside strategy S3 produces large test suites, containing duplicate test cases, which increases the time needed to execute them on the implementations. Strategy S4 reduces the number of test cases, but takes more time to generate them. Also, without the cut of the search tree, they are only feasible for small models. Enabling the cut of the search tree, lowers both run-time and size of the generated test suites. Since the strategies S3 and S4 loose their advantages, strategy S5 becomes a good trade-off. However, the effectiveness of the generated test suites in terms of killing faulty implementations suffers.

6 Case Study: Wheel Loader

Parts of this chapter have been submitted to but not yet published by the Journal of Software Testing Verification, Validation and Reliability (STVR) [Aic+].

The second case study deals with a wheel loader. The requirements have been provided by RE:LAB, an industry partner within the MOGENTES project.

The wheel loader consists of a bucket, which is connected to a tractor. It is controlled by a human, using a joystick to move both the bucket and the bucket arm. The core component of the system-under-test is an electronic control unit (ECU), which is connected to an ISOBUS network. ISOBUS is an extension to the widely-known CAN bus technology for the automotive industry. Besides of the ECU, in this network there is the joystick used to control the bucket and a TFT display, also referred to as *Virtual Terminal*, which shows the status of the system and can be mounted in the cabin. Also, the ECU controls the actuators of the bucket and the bucket arm. This is done by setting the currents of two electromagnets, of which each controls a "hydraulic distributing valve". The ECU itself has been implemented on a Freescale i.MX35 processor running Linux. For demonstration purposes within the MOGENTES project, a Lego model of the wheel loader has been built by RE:Lab and the ECU, the joystick and the TFT display have been connected. Figure 6.1 shows the setup of this demonstration, which has been performed by RE:Lab within one MOGENTES meeting.

6.1 Requirements

Unlike the requirements of the car alarm system, which could be condensed to only three core requirements, the requirements of the wheel loader cover a broad range of different aspects. These aspects are input handling, error management, providing suitable values fulfilling the physical constraints, and general timing related properties.

6.1.1 Input Handling

The wheel loader is controlled by an eight-way joystick. That means that the joystick can be moved in the cardinal directions as well as in the diagonals. The cardinal directions are along the two axes. The deflection in Axis 1 controls the position of the bucket arm, while the deflection in Axis 2 controls the rotation of the bucket. When the joystick is moved backward (Axis 1), the bucket arm moves towards the top, when the joystick is moved forward, the bucket arm moves towards the bottom. When the joystick is moved

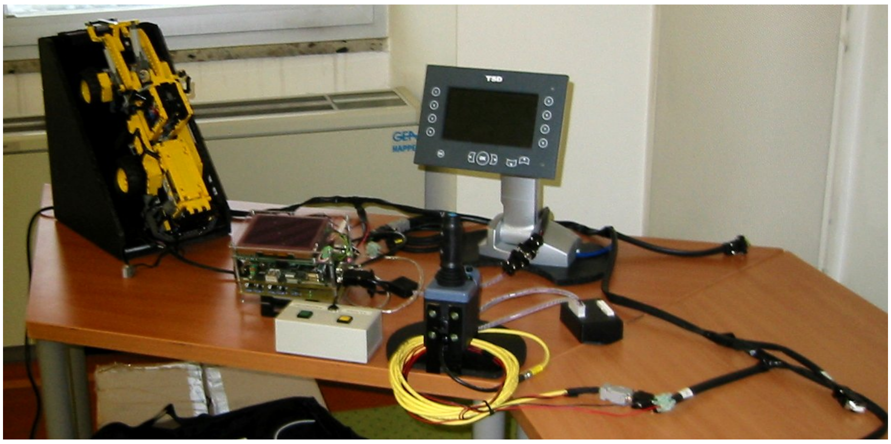


Figure 6.1: Demonstration setup of a wheel loader Lego model

left (Axis 2), the bucket rotates towards the top, when the joystick is moved right, the bucket rotates towards the bottom.

The deflection of the joystick is sent using ISOBUS messages, containing two Integer values in a range between 0 and 65355. This range is divided into a “valid” range and an “error” range. The valid range is from 0 to 64255 and its values represent an actual joystick deflection, while values from 64256 to 653555 are in the “error” range used to communicate problems like a stuck switch or a broken wire.

The deflection value of the neutral position of the joystick is 32127 on both axes. In the requirements document this position is also referred to as “rest position”.

6.1.2 Error Handling

For error handling there is an extended state machine defined in the requirements document as depicted in Figure 6.2. The ECU reacts to faults reported by the joystick by switching the state within this state machine. The state machine uses three variables (`un_valid_data_counter`, `correction_counter`, and `valid_data_counter`) and three constants (`K`, `M`, and `N`), which can be set by the original equipment manufacturer (OEM). The initial state is called *NO_ERROR_DETECTED*. As soon as the first joystick input representing an error value is received, the system is switched in the *FILTERING* state. In this state, the system functionality is still enabled, but further faults reported by the joystick are counted. If the number of reported faults exceeds the threshold `M`, the system functionality is disabled, and the system state is changed to *CONFIRMATION* or *STOP*. The first `K - 1` times, the system goes into the *CONFIRMATION* state, from which it can still recover, the $\texttt{K}^{th}$ time it goes to the *STOP* state, where the system functionality is disabled until the system is restarted. In the *CONFIRMATION* state the ECU waits for inputs of the joystick using valid values. If this happens, the state

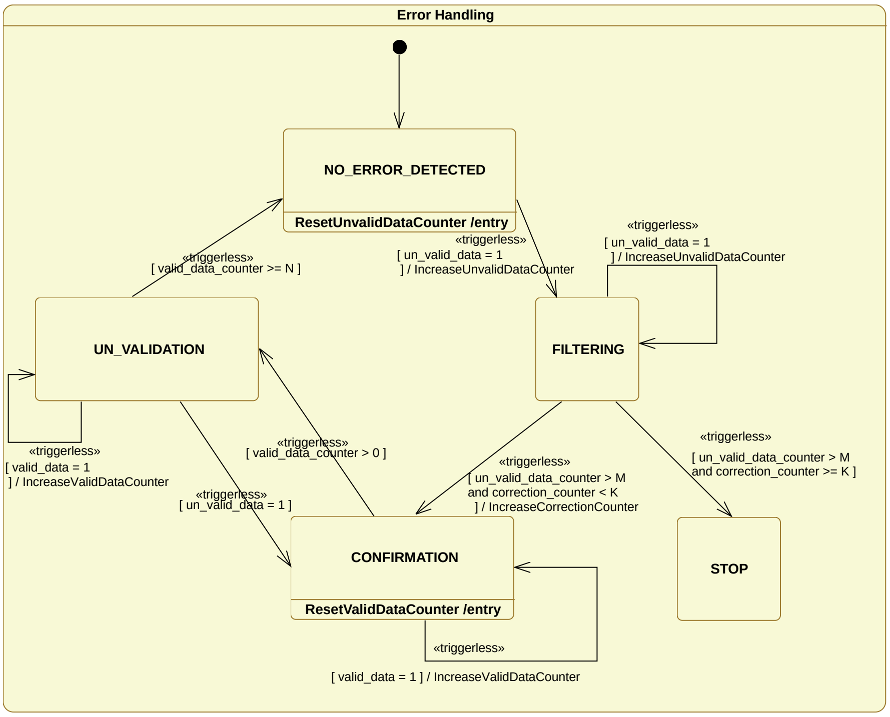

Error Handling
NO_ERROR_DETECTED
ResetUnvalidDataCounter /entry
«triggerless»
[valid_data_counter >= N]
«triggerless»
[un_valid_data = 1
] / IncreaseUnvalidDataCounter
«triggerless»
[un_valid_data = 1
] / IncreaseUnvalidDataCounter
UN_VALIDATION
FILTERING
«triggerless»
[valid_data = 1
] / IncreaseValidDataCounter
«triggerless»
[valid_data_counter > 0]
«triggerless»
[un_valid_data = 1]
«triggerless»
[un_valid_data_counter > M
and correction_counter < K
] / IncreaseCorrectionCounter
«triggerless»
[un_valid_data_counter > M
and correction_counter >= K]
CONFIRMATION
ResetValidDataCounter /entry
STOP
«triggerless»
[valid_data = 1] / IncreaseValidDataCounter

Figure 6.2: Error handling state machine of the ECU of the wheel loader

changes to *UN_ VALIDATION*, where further valid deflection values are expected. If at least `N` times in series a valid deflection valid has been received, the system traverses back in the *NO_ ERROR_ DETECTED* state, where the system functionality is enabled again.

6.1.3 Virtual Terminal

The virtual terminal is the interface to the TFT display mounted in the cabin of the wheel loader. The display shows the current state of the system including the current state of the error handling, the current deflection of the joystick as well as the direction the bucket and the bucket arm are moving, which is an abstraction of the outputs of the ECU to the electromagnets. Because of that, the virtual terminal is divided into three areas. For each area there is a predefined set of possible graphics. There are three possible pictures showing the error handling state, allowing to distinguish an “OK Status” for the states, in which the system functionality is enabled, “Error” for the states, in which the functionality is disabled but the system can still recover, and “Critical Error” for the state, from which the system does not recover but has to be restarted. Additionally, there are nine graphics showing the directions in which bucket and bucket arm are moving and a total of 81 graphics showing the current deflection of the joystick.

Whenever a graphic changes, the ECU has to send an ISOBUS message in order to trigger the update of the TFT display.

6.1.4 Electromagnets

In order to actually cause movements of the bucket and its arm, the ECU has to set the current of two electromagnets. One electromagnet is responsible for the bucket and one is responsible for the bucket arm. Each of them has two ports. One port is responsible for the movement towards the top, the other is responsible for the movement towards the bottom. The current of each port ranges from 0A (not powered) to 0.8A (fully powered). The currents must not jump outright from 0A to 0.8A and vice versa. Instead the ECU has to perform a so-called "ramp mode" in which the values change slowly. The minimum time needed for a change from 0A to 0.8A can be defined by the OEM and can be different from the time needed for a change from 0.8A to 0A. Also the durations can be different between the movement of the bucket and the movement of the bucket arm. So the deflections of the joystick are not directly mapped to the actual output currents, but instead they are mapped to target values, and the actual currents are changed slowly until they match the target value. The mapping between the deflection values and the target output values is defined as follows: deflection values in an ϵ-neighborhood are mapped to 0A, the extreme deflections of 0 resp. 64255 are mapped to 0.8A and all other values are mapped linearly.

6.1.5 Timing Properties

The requirements document implicitly includes two aspects of timing properties: The first aspect is that outputs to the electromagnet have to be provided by a fixed rate defined by the OEM (for example each 100 ms), whereas the joystick has a maximum transmission rate of 5 Hz which means that new deflection values can occur only every 200 ms, and it can even take up to one second. The second aspect of timing properties is that each component in the ISOBUS network has to send a so-called heart-beat message to show that it is still alive and if there is a time-out for one of them, the ECU has to stop the system functionality and traverse into the *STOP* state of the error management.

The heart-beat message of the joystick is called "Auxiliary input maintenance message" and has to oocur at least every 300 ms, the heart-beat message of the virtual terminal is called "VT Status Message" and the ECU itself has to send a heart-beat message called "Working Set Maintenance Message" every second.

6.1.6 ISOBUS initialization

Another set of requirements concerns the initialization of the ISOBUS. First the ECU claims its address and broadcasts it to the network. If for 250 ms, no other component vetoes, the initialization is completed. Otherwise another component claims the same address. In this case, a protocol ensures that the component with the smaller numerical value in its name wins and the other component has to choose another address.

6.2 Test Models

6.2.1 Preexisting Models

Within the MOGENTES project, a first test model has been created by AIT. The class diagram is divided into three parts and shown in Figure 6.3, Figure 6.4, and Figure 6.5.

As can be seen in Figure 6.3 the parameters, that can be set by the OEM are stored as constants in an own class called “Parameter Set”. Here the constants `K`, `M`, and `N` are defined for the error handling, described in Section 6.1.1. The property `cycleTime` denotes the output rate of the ECU towards the electromagnets. The properties `maxStepUpBucket` etc. are used to parameterize the ramp mode, which is described in Section 6.1.4. Instead of defining the time needed for a change from 0 A to 0.8 A, the maximum current change per update cycle is stored.

The ECU itself provides the following properties:

- the variables defined by the error handling
- the target value of each electromagnet current as well as the current values needed for the next update cycle (`RequestedOut`)
- the current graphics shown on the TFT display
- the ISOBUS address (`SA`)
- the dummy element `msg` used to access the parameters of the input signals
- the address within ISOBUS network

Additionally the class defines private operations, in which AGSL code is swapped out that is used more than once in order to avoid redundancy.

Also in this class diagram the data types are defined. For the graphics used on the TFT display an enumeration type is used, all other data types are restricted Integers. Note, that the ranges used for the joystick deflection are too wide (0 to 65355) for the enumerative approach of *Ulysses*, since it would lead to 4271276025 different input events for each step, where an input occurs.

Because of that, the input values have to be restricted more rigorously, leading to a model with less possible different input labels. Such a model is an abstraction in terms of under-specification as described in Section 4.1.

Figure 6.4 depicts the test interface of the model. The system boundaries define the ECU as system_under_test and the electromagnets and the virtual terminal as environment. Messages from the ECU to the environment occur as output events in the LTS and therefor in the test case. For setting the current on the electromagnet, a public operation with four Integer parameters is defined. Outputs from the ECU to the virtual terminal via the ISOBUS network are defined as signal reception. The input messages to the

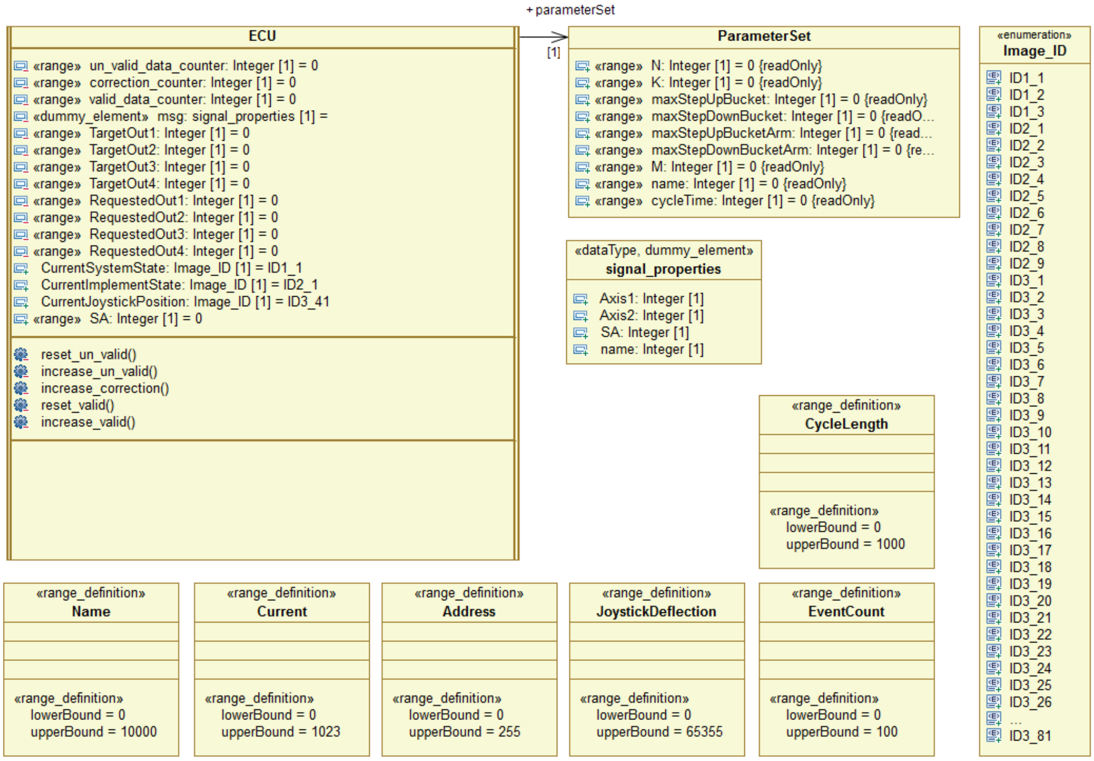

+parameterSet
[1]
ECU
«range» un_valid_data_counter: Integer [1] = 0
«range» correction_counter: Integer [1] = 0
«range» valid_data_counter: Integer [1] = 0
«dummy_element» msg: signal_properties [1] =
«range» TargetOut1: Integer [1] = 0
«range» TargetOut2: Integer [1] = 0
«range» TargetOut3: Integer [1] = 0
«range» TargetOut4: Integer [1] = 0
«range» RequestedOut1: Integer [1] = 0
«range» RequestedOut2: Integer [1] = 0
«range» RequestedOut3: Integer [1] = 0
«range» RequestedOut4: Integer [1] = 0
CurrentSystemState: Image_ID [1] = ID1_1
CurrentImplementState: Image_ID [1] = ID2_1
CurrentJoystickPosition: Image_ID [1] = ID3_41
«range» SA: Integer [1] = 0
reset_un_valid()
increase_un_valid()
increase_correction()
reset_valid()
increase_valid()
ParameterSet
«range» N: Integer [1] = 0 {readOnly}
«range» K: Integer [1] = 0 {readOnly}
«range» maxStepUpBucket: Integer [1] = 0 {readOnly}
«range» maxStepDownBucket: Integer [1] = 0 {readO...
«range» maxStepUpBucketArm: Integer [1] = 0 {read...
«range» maxStepDownBucketArm: Integer [1] = 0 {re...
«range» M: Integer [1] = 0 {readOnly}
«range» name: Integer [1] = 0 {readOnly}
«range» cycleTime: Integer [1] = 0 {readOnly}
«dataType, dummy_element»
signal_properties
Axis1: Integer [1]
Axis2: Integer [1]
SA: Integer [1]
name: Integer [1]
«range_definition»
CycleLength
«range_definition»
lowerBound = 0
upperBound = 1000
«enumeration»
Image_ID
ID1_1
ID1_2
ID1_3
ID2_1
ID2_2
ID2_3
ID2_4
ID2_5
ID2_6
ID2_7
ID2_8
ID2_9
ID3_1
ID3_2
ID3_3
ID3_4
ID3_5
ID3_6
ID3_7
ID3_8
ID3_9
ID3_10
ID3_11
ID3_12
ID3_13
ID3_14
ID3_15
ID3_16
ID3_17
ID3_18
ID3_19
ID3_20
ID3_21
ID3_22
ID3_23
ID3_24
ID3_25
ID3_26
...
ID3_81
«range_definition»
Name
«range_definition»
lowerBound = 0
upperBound = 10000
«range_definition»
Current
«range_definition»
lowerBound = 0
upperBound = 1023
«range_definition»
Address
«range_definition»
lowerBound = 0
upperBound = 255
«range_definition»
JoystickDeflection
«range_definition»
lowerBound = 0
upperBound = 65355
«range_definition»
EventCount
«range_definition»
lowerBound = 0
upperBound = 100

Figure 6.3: Class diagram ECU of the wheel loader

ECU also come from the ISOBUS network and are also defined as signal receptions. The signals are defined in an extra class diagram as depicted in Figure 6.5.

Figure 6.6 depicts the version of the state machine of the whole wheel loader ECU, which had been used, before this thesis has started. The initial state is called “AddressClaim” which is a relict of the ISOBUS initialization phase, which had been originally modelled in the same test model. It soon turned out, that the ISOBUS initialization had to be modelled in an own partial model, so it has been already deactivated by removing the transition from the sub-state “Wait” to the sub-state “Rcv”. So the only thing left is the sending of the address by the ECU as entry action and after 250 ms the system enters the “Initialized” state. This state is a super-state containing six orthogonal regions. The first region models the error handling which is described in Section 6.1.2. The only difference is that the states in which the system functionality is enabled are grouped in a super-state “Active” and the states in which the system functionality is currently disabled but can be recovered are grouped in a super-state “Inactive”. The second region is responsible for processing the inputs coming from the joystick. In the state “Receiving” an “AuxiliaryInputStatusMessage” is received with the joystick deflection as parameters and then processed. If one of the values is in the error range, the corresponding variables of the error handling mechanism are updated. Otherwise a new target value for

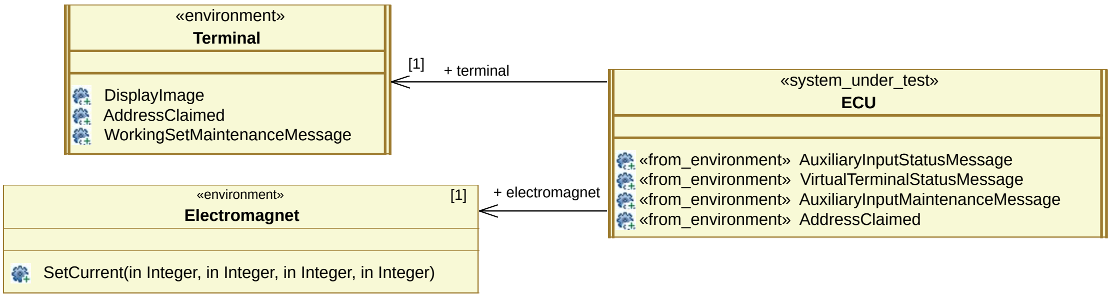

«environment»
Terminal
DisplayImage
AddressClaimed
WorkingSetMaintenanceMessage
[1]
+ terminal
«system_under_test»
ECU
«from_environment» AuxiliaryInputStatusMessage
«from_environment» VirtualTerminalStatusMessage
«from_environment» AuxiliaryInputMaintenanceMessage
«from_environment» AddressClaimed
«environment»
Electromagnet
[1]
+ electromagnet
SetCurrent(in Integer, in Integer, in Integer, in Integer)

Figure 6.4: Test interface of the wheel loader

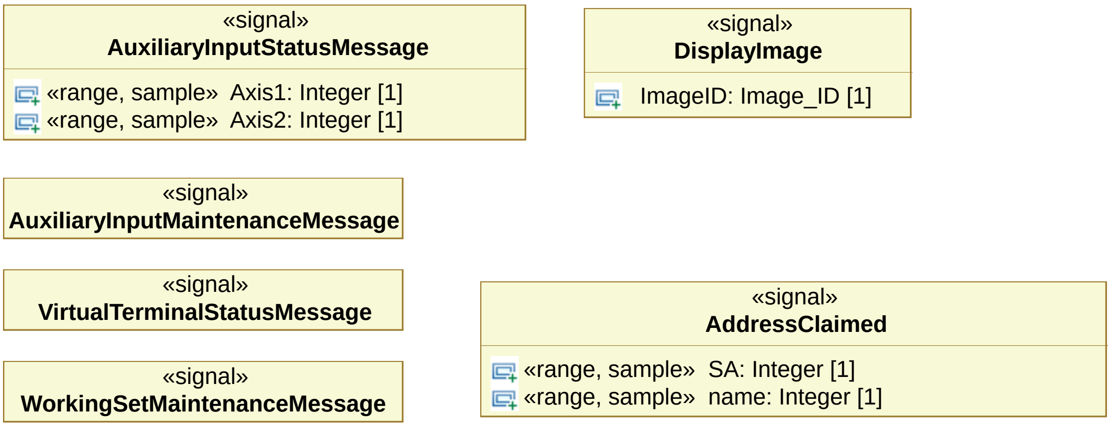

«signal»
AuxiliaryInputStatusMessage
«range, sample» Axis1: Integer [1]
«range, sample» Axis2: Integer [1]
«signal»
DisplayImage
ImageID: Image_ID [1]
«signal»
AuxiliaryInputMaintenanceMessage
«signal»
VirtualTerminalStatusMessage
«signal»
WorkingSetMaintenanceMessage
«signal»
AddressClaimed
«range, sample» SA: Integer [1]
«range, sample» name: Integer [1]

Figure 6.5: Class diagram ISOBUS messages of the wheel loader

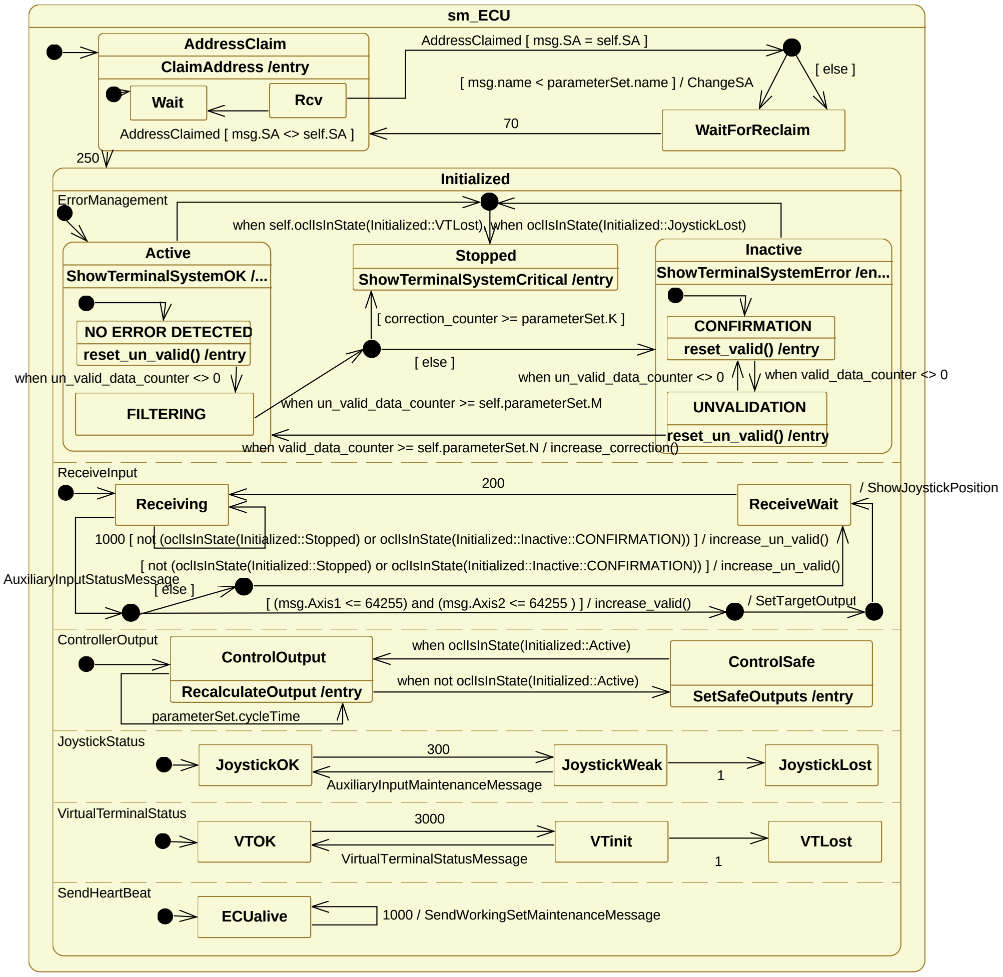

sm_ECU
AddressClaim
ClaimAddress /entry
Wait
Rcv
AddressClaimed [msg.SA <> self.SA]
250
AddressClaimed [msg.SA = self.SA]
[msg.name < parameterSet.name] / ChangeSA
[else]
70
WaitForReclaim
Initialized
ErrorManagement
when self.oclIsInState(Initialized::VTLost)
when oclIsInState(Initialized::JoystickLost)
Active
ShowTerminalSystemOK /...
Stopped
ShowTerminalSystemCritical /entry
Inactive
ShowTerminalSystemError /en...
NO ERROR DETECTED
reset_un_valid() /entry
when un_valid_data_counter <> 0
FILTERING
[correction_counter >= parameterSet.K]
[else]
when un_valid_data_counter >= self.parameterSet.M
CONFIRMATION
reset_valid() /entry
when un_valid_data_counter <> 0
when valid_data_counter <> 0
UNVALIDATION
reset_un_valid() /entry
when valid_data_counter >= self.parameterSet.N / increase_correction()
ReceiveInput
Receiving
200
ReceiveWait
/ ShowJoystickPosition
1000 [not (oclIsInState(Initialized::Stopped) or oclIsInState(Initialized::Inactive::CONFIRMATION))] / increase_un_valid()
[not (oclIsInState(Initialized::Stopped) or oclIsInState(Initialized::Inactive::CONFIRMATION))] / increase_un_valid()
AuxiliaryInputStatusMessage
[else]
[(msg.Axis1 <= 64255) and (msg.Axis2 <= 64255)] / increase_valid()
/ SetTargetOutput
ControllerOutput
ControlOutput
RecalculateOutput /entry
when oclIsInState(Initialized::Active)
when not oclIsInState(Initialized::Active)
ControlSafe
SetSafeOutputs /entry
parameterSet.cycleTime
JoystickStatus
JoystickOK
300
JoystickWeak
AuxiliaryInputMaintenanceMessage
1
JoystickLost
VirtualTerminalStatus
VTOK
3000
VTinit
VirtualTerminalStatusMessage
1
VTLost
SendHeartBeat
ECUalive
1000 / SendWorkingSetMaintenanceMessage

Figure 6.6: State machine of the wheel loader

the electromagnet currents is calculated and the graphic depicting the current deflection is sent to the virtual terminal. This leads to the state “ReceiveWait” in which the system stays for 200 ms, which corresponds to the maximum transmission rate for the joystick of 5 Hz, before returning in the “Receiving” state. In the third region, the actual currents are calculated and sent to the electromagnet. This is done as entry action of the state “ControlOutput”, which has a self-loop triggered by a time trigger waiting as long as defined by the constant `cycleTime`. If the error handling changes to the state, in which the system functionality is disabled, the system changes to the “ControlSafe” state. The last three regions are responsible for the time out events. If the joystick does not send the “AuxiliaryInputMaintenanceMessage” for more than 300 ms or the virtual terminal does not send the “VirtualTerminalStatusMessage” for more than 3 seconds, the error handling stops the functionality of the system. The ECU itself has to send the “WorkingSetMaintainenceMessage” once per second.

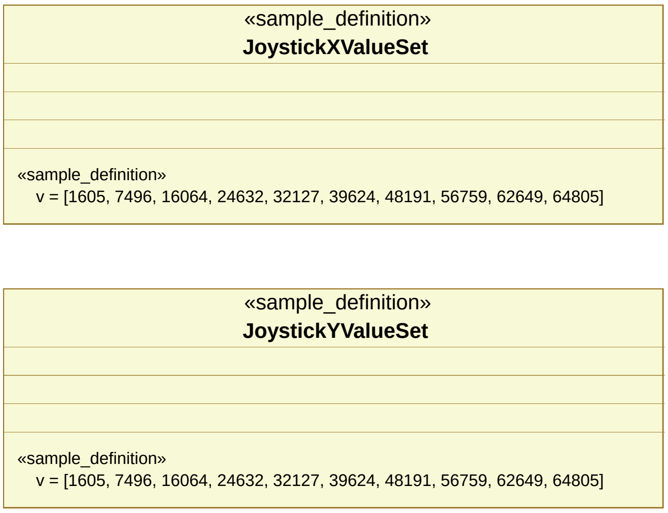

«sample_definition»
JoystickXValueSet
«sample_definition»
v = [1605, 7496, 16064, 24632, 32127, 39624, 48191, 56759, 62649, 64805]
«sample_definition»
JoystickYValueSet
«sample_definition»
v = [1605, 7496, 16064, 24632, 32127, 39624, 48191, 56759, 62649, 64805]

Figure 6.7: Equivalence class partitioning sample definition model “EQC”

Partial Models To a limited extent, the idea to use partial models has already been incorporated within the MOGENTES project. As already mentioned, the idea to use one model for both ISOBUS initialization phase and the rest of the requirements has been given up immediately. Another point are the used values for the joystick deflection. Having two Integer parameters with a range from 0 to 65355 results in 2^{32} possible combinations. Using the enumerative approach of *Ulysses*, these are by far too many. While a parameter range from 0 to 300 is acceptable for internal actions, parameters used in input and output events have to be restricted more rigorously.

Instead of using a continuous range of values, it is necessary to do a so-called equivalence class partitioning. This is a common technique in software testing, where input data is divided into several classes and input data within the same class are assumed to be equivalent in terms of revealing faults in a system. Then for each equivalent class one value can be chosen arbitrarily and used to represent the whole class of test data.

In the test model, the chosen values can be enumerated explicitly using the stereotype «sample_definition» as described in Section 3.1.2. Within the MOGENTES project two partial models have been derived from the test models by using different equivalence class partitions. They have been referred to as *EQC* resp. *X_error*.

The term *EQC* is an acronym for “equivalence classes”. The partitions are based on the graphics, sent to the virtual terminal. That means that for each graphic depicting a particular joystick position one value is chosen. Additionally to these nine values per axis, there is a value from the error range, which is used by the joystick to report a fault, in order to be able to trigger the error handling. Since this value is also used for both axes, there is a total of 10 values per axis, leading to 100 possible combinations and therefore 100 different input events in the generated LTS. Figure 6.7 shows the sample definition.

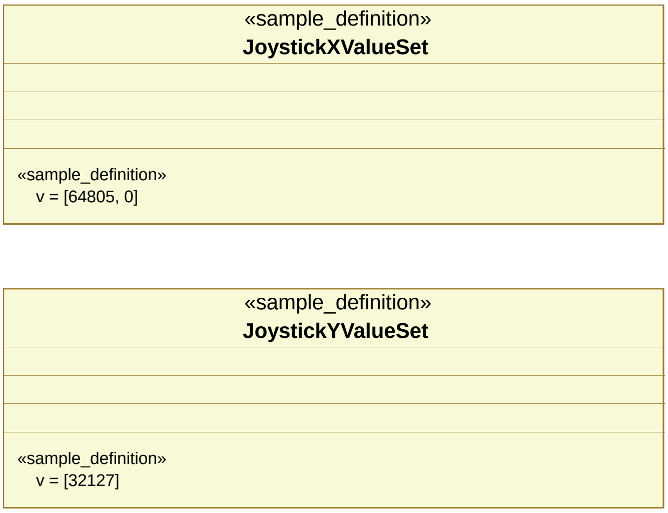


Figure 6.8: Equivalence class partitioning sample definition model “X_error”

This test model captures the whole system functionality except for the ISOBUS initialization. The random test case generation approach of *Ulysses* can be used to generate test cases covering error handling as well as the calculation of the currents for the electromagnets as well as sending the correct pictures to the TFT display. However, the mutation-based approach suffers from the so-called “state space explosion”. In each step, where an input from the joystick occurs, execution branches 100 times. Because of that performing an *ioco* check is only feasible when small values for the search depth are used. Exploring the system up to the point, where the first joystick input is processed and all output events are considered, took about three hours.

Figure 6.8 shows the sample definition for another equivalence class partitioning, which is based solely on the error handling. This model is also referred to as “X_error”. Instead of considering ten values per axis, in this model the Y-axis uses a fixed value “32127” which corresponds to the neutral position, while the X-axis uses the value “0” which corresponds to the left-most joystick deflection and the value “64805” which is in the error range and therefore corresponds to a fault reported by the joystick.

Instead of 100 choices each time, a joystick input occurs, only two choices have to be considered. Because of that, it is feasible to perform an *ioco* check up to a depth, where the behavior of the error handling can be observed. So *X_error* can be seen as first partial model for the wheel loader case study.

One main characteristic of the underlying modeling style is the extensive use of orthogonal regions. Note, that in order to generate comprehensive tests of the error handling, also processing the joystick input and generating the output to the electromagnet have to be modeled. Within this thesis it has been tried to provide an alternative version using the multiple classes modeling style, as presented in Section 5.2.3. Unfortunately, due to some limitations within the UML-to-OOAS transformation this has not been successful.

Unlike the models of the car alarm system, where inputs from the environment did not contain any parameters, when modeling the ECU of the wheel loader, there are parameters. Because of some issues when trying to access the information received as parameter of an input action, there would have been the need for additional workarounds. Since these workarounds would have caused additional overhead within the UML-to-OOAS transformation, obtaining a feasible model using the "multiple classes" modeling style did not look promising.

6.2.2 Improvements

General Bottlenecks A first attempt to improve the test model is to identify bottle necks. One of them is the lapse of time, which on the OOAS level is modelled using an internal action called *after(t)*. This action is enumerated t times, where t is the maximum time value used in a time trigger of the test model. This means that in each step, where an input event can occur, there are t calls of the *after(t)* action non-deterministically composed to each of the input events. Experiments by Aichernig and Jöbstl [AJ12] have shown, that *Ulysses* is very sensitive about the range of these parameters. Compared to the test model of the car alarm system, in which the highest time unit has been 300 time units, the original test model of the wheel loader uses time events up to 3000 time units. While a range from 0 to 300 for the internal after event works fine with *Ulysses*, a range from 0 to 3000 can render the enumeration to a bottleneck of the exploration. To avoid this issue, it is better to change the time unit. When defining one time unit in the test model to correspond to 10 ms in reality, values used in the model are limited by 300, which is known to work.

Another optimization is to remove all unused transitions in the ISOBUS initialization phase. Even though the transitions are not reachable, they are still translated into the object-oriented action system as internal actions and called in the iterative loop of the corresponding action system.

New Partial Model Since the partial model for testing the error handling has been already feasible at the start of this thesis, most efforts have been put on the second partial model. Two reasons can be identified, why the "EQC" model is so complex: firstly, 100 different choices of possible joystick deflection values are too many. Since runtime grows exponentially to the number of choices, choices have to be restricted more rigorously. Secondly, the model consists of six orthogonal regions, covering also the error handling, which is already tested using the other partial model.

Figure 6.10 shows the state machine of the partial model that has been created within this thesis. When modeling the system without error handling, all the other orthogonal regions can be merged. In this case, the test model only specifies joystick input that represents a valid joystick deflection values. Only those parts of error handling have to be considered, which are responsible for reacting to timeouts caused by missing keep alive messages. Note, that the guards are hidden because of readability reasons.

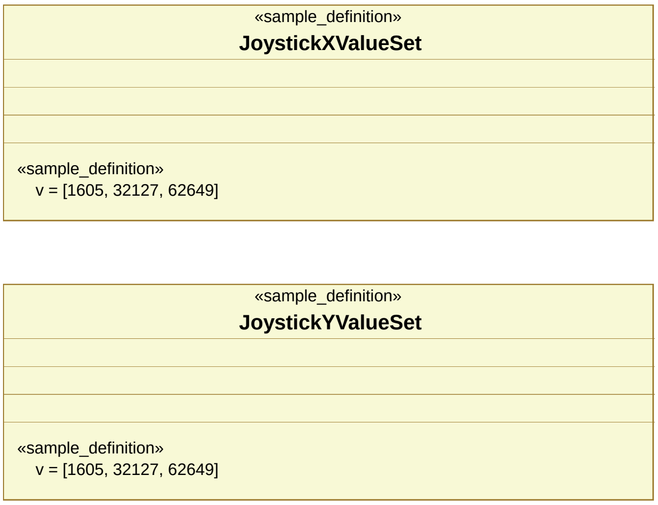


Figure 6.9: Equivalence class partitioning sample definition model "Extremes"

Figure 6.9 shows the corresponding sample definition. The equivalence class partitioning has been done on extreme values. They include the four cardinal directions (left, right, up, down), the four corners as well as the neutral position. Because of that it is referred to as model "Extremes".

To make the state machine as simple as possible, only one region has been used. Also, each opaque behavior is linked to a transition and no entry or exit action is used. The initial state is "q0", where the startup phase begins. The first output events occur until the system enters the state "q5". In this state the first input from the joystick can occur, leading to state "q6", or another 100 ms (10 time units) pass without an input, leading directly to state "q9". Here also the update loop begins. Note, that it is only possible to merge all six orthogonal regions of the original model into one big loop, if some input events are left out, in order to synchronize the input and output events. This can be done, since it leads to a more abstract model, leaving the missing inputs under-specified.

In this model, each iteration of the loop lasts 300 ms (30 time units) and includes at most one joystick input, while the original model allowed for an joystick input every 200 ms. The states "q11", "q14", and "q20" contain a self-loop with an effect of sending the heart-beat message of the ECU. To ensure, that this is done exactly once every second, the current time is stored in a class property called `WorkingSetClock` which stores the time since the last heart-beat messages in ms. The self-loop has a guard with the condition `WorkingSetClock = 1000` and the other outgoing transitions have a guard with the condition `WorkingSetClock < 1000`. From state "q20" there is a transition triggered by a time event of one time unit. This transition is taken, if the heart-beat message of the

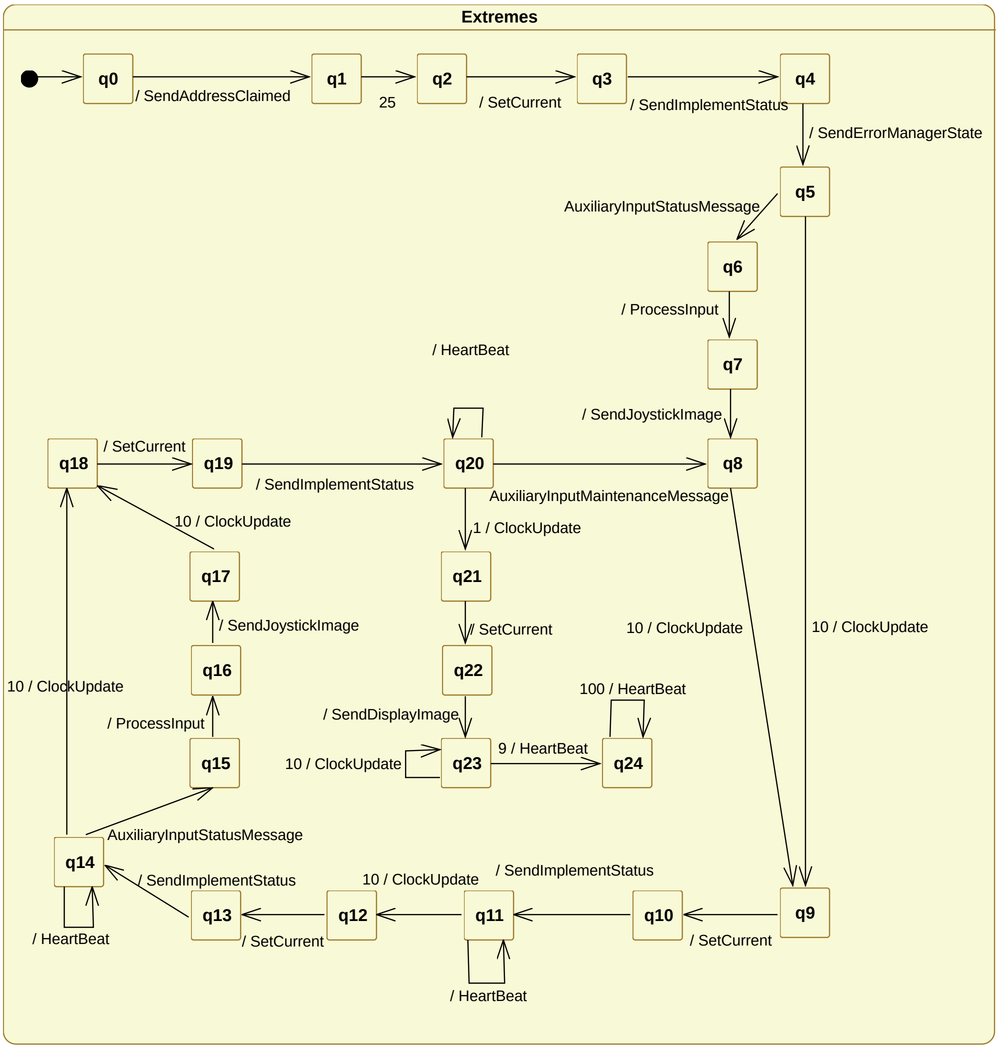


Figure 6.10: State machine, model “Extremes”

joystick is missed, leading to a timeout. In this case the system functionality is disabled and only the own heart-beat message is sent every second (100 time units).

The underlying modeling style resembles to the “Simple State Machine” modeling style, presented in Section 5.2.1. However, there is some information stored in the properties, so technically it is a combination of the “Simple State Machine” and the “Action System Like” modeling style.

Sample Test Case Figure 6.11 shows a sample test case for the wheel loader ECU. Output events from the ECU are marked with the prefix “obs”, the input event is marked with the prefix “ctr”. The first parameter denotes the time that has passed before the event occurs. Data using an enumeration type like the id of the pictures sent to the TFT display is represented by Integers. The mapping is done by the tool *Argos* and has to be considered when writing the test driver.

The test case describes the following scenario: Immediately after started (0 time units) the ECU provides its name (0) and claims its ISOBUS address (4). This is modeled in the effect named “SendAddressClaimed”. Then 250 ms afterwards (25 time units) it sets the currents of each port of each electromagnet to “0” (effect “SetCurrent”). Immediately after that (0 time units) the ECU sends the picture “3”, denoting that the bucket is currently not moving (effect “SendImplementStatus”) and finally picture “0”, denoting that it is in a healthy state (effect “SendErrorManagerState”). Without waiting (0 time units) the joystick sends its current deflection (right most on X-axis, no deflection on Y-axis) to the ECU (signal event “AuxiliaryInputStatusMessage”). The desired behavior is a rotation towards the bottom, but no movement of the bucket arm. The ECU has to send picture “56”, corresponding to the joystick position immediately (effect “SendJoystickImage”) and set the current of Port A of electromagnet 1 to 50 mA after 100 ms (effect “SetCurrent”).

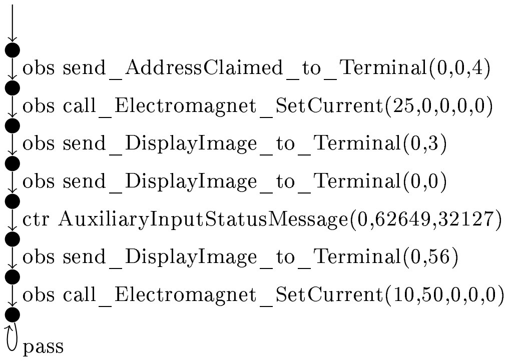

obs send_AddressClaimed_to_Terminal(0,0,4)
obs call_Electromagnet_SetCurrent(25,0,0,0,0)
obs send_DisplayImage_to_Terminal(0,3)
obs send_DisplayImage_to_Terminal(0,0)
ctr AuxiliaryInputStatusMessage(0,62649,32127)
obs send_DisplayImage_to_Terminal(0,56)
obs call_Electromagnet_SetCurrent(10,50,0,0,0)
pass

Figure 6.11: A sample test case for the wheel loader case study

6.3 Experiments

The aim of the experiments within this case study has been to repeat the comparison between the different killing strategies, described in Section 3.4.2. For the car alarm system this has already been published [Aic+11a].

As test model for testing the error handling the *X_ error* model and for testing the positive behavior the *Extremes* model have been chosen.

6.3.1 Mutation

To make the experiment feasible parameters had to be chosen, in order to keep the needed time per model and killing strategy below one week. This can be done by restricting the search depth or by selecting the mutation operators. Applying all mutation operators on the two partial models would result in 418 mutants for the *X_ error* model and 2004 mutants for the *Extremes* model. The reason, why the latter model yields so many mutants is that this model has more redundancy. To avoid multiple orthogonal regions, which lead to more complex object-oriented action system, some AGSL code fragments have been duplicated. For instance the code responsible for calculating the currents for the electromagnet occur only once in the *X_ error* model but three times in the *Extremes* model.

Another observation is that many mutations occur in the startup and initialization phase. Test cases yielded by these mutations would be included in most other test cases, which test for more subtle faults. So instead of using all mutations, only the following three mutation operators have been chosen:

1. Mutating Transition Time Trigger Events.
2. Mutating Transition OCL expressions.
3. Mutating Transition AGSL expressions.

These mutation operators replace all literal Integer values by other values, for this experiment the value increased by one has been chosen. Together these mutation operators capture a fault-model of so-called off-by-one errors.

Additionally mutants have been filtered out, where the mutation regards the mapping between the joystick deflection and the picture sent to the TFT display. Since the equivalence class partitioning is not based on the joystick pictures anymore, this can be considered as so-called “dead code”. That means, that the relevant code can not be reached using this input values. To decide, which picture is displayed on the TFT display, a nested conditional assignment is used. Using the equivalence class partitioning containing only three values per axis, most branches of the statement can not be reached and therefore a mutation in there can not be found.

Finally, this results in a total of 58 mutants for the *X_ error* model and 192 mutants for the *Extremes* model.

Model Metrics	X_err	Extr
Max. depth for *ioco* check	35	25
Mutants [#]	58	192
Equivalent mutants [#]	35	106
Mutants causing a livelock	0	8

Table 6.1: Important characteristics for both test models of the wheel loader model.

6.3.2 Test Case Generation

Setup As described in Chapter 3 the *MoMuT* toolchain consists of several loosely connected separate tools.

For these experiments the creation resp. adaption of the UML models with *Papyrus* as well as mutating with *UMMU* and translating the models into action systems using *Argos* has been done using a laptop computer with one 2.0 GHz quad-core processor and 4 GB RAM.

The core part of the experiments, the generation of test cases using *Ulysses*, has been conducted on a computer with two 2.5 GHz quad-core processors and 32 GB RAM running a 64-bit Linux as operating system. This computer is also used by other work-groups of the institute, but five parallel processes have been granted to these particular experiments and up to 25 GB RAM could be used.

For parameterizing *Ulysses* to run each test model with each killing strategy, shell scripts have been used.

Results All five strategies described in Section 3.4.2 (S3 – S7) have been applied to both test models. The mutation-based approaches (S3 – S6) have been used with the cut of the search tree enabled. Strategy S7, which does a random test case generation, has been applied three times. Also, the strategy S6, which uses random test cases generated by strategy S7 as initial test suite, has been applied three times. The reported values are therefor mean values of the three runs.

It has also been attempted to apply the same strategies with the cut of the search tree disabled. However, this attempt had to be cancelled, since within two months, not even processing the first test model could be completed by any of the killing strategies with the cut of the search tree disabled. It already took 24 days just to process the first model mutant of the *X_ error* test model with strategy S3, which yielded alone 1190 test cases.

Table 6.1 lists the metrics of the models which are independent of the strategy: for the *Extremes* model a search depth of 25 and for the *X_ error* model a search depth of 35 has been used. Up to these exploration depths, 106 of the 192 mutants of the *Extremes* model and 35 of the 58 mutants of the *X_ error* model have been equivalent, that means, they are input-output conform to the original model with respect to the given exploration

Strategy	S3	S4	S5	S6	S7
X_error	31	17	14	13.3	3
Extremes	356	150	60	28.3	3

Table 6.2: Number of generated test cases per test model and strategy

Strategy	S3		S4		S5		S6	
Model	X_err	Extr	X_err	Extr	X_err	Extr	X_err	Extr
Avg. t for equivalent mutant	161.8	35	162.3	35	162.7	36	156.8	36.8
Total t for equivalent mutants	5663	3711	5679	3709	5696	3812	5487	3849
Avg. t for mutant causing livelock	0	6.2	0	6.2	0	7.2	0	5.6
Total t for mutants causing livelock	0	50	0	50	0	58	0	40
Mutants triggering TC [#]	23	78	9	41	6	23	5.3	9.3
Avg. t for mutant triggering TC	7.3	2.1	19.6	5.9	27.7	3.4	36.2	6.8
Total t for mutants triggering TC	168	167	176	241	166	79	181	410
Mutants killed by existing TC [#]	–	–	14	37	17	55	17.6	71
Avg. t for mutant killed by TC	–	–	0.1	1.7	0.1	0.3	0.4	0.7
Total t for mutants killed by TC	–	–	2	63	1	15	25	48
Total t for TC gen	5831	3928	5857	4063	5863	3964	5693	4347

Table 6.3: Run-times t in minutes for generating test cases per strategy.

depths. Note, that they are not necessarily really equivalent as behavioral differences might occur deeper in the state space. On average processing a mutant up to this point took 35 minutes.

Since it might happen that mutated specifications run into infinite loops of internal actions, a so-called *livelock*, a timeout of 5 minutes for each step between two visible actions during test case generation has been set. If this timeout is reached, the test case generation fails and no test case is generated. Eight of the *Extremes* mutants caused a livelock, but none of *X_error*.

The main difference among the fault-based test case generation strategies S3–S6 lies in the number of generated test cases. Table 6.2 lists the number of test cases for each application of a strategy for both test models. Compliant with the results of the car alarm system case study the number of generated test cases decreases as the strategy gets more sophisticated.

For the *X_error* model the number of generated test cases ranges from 31 using the strategy S3 to 14 using strategy S5 and an average of 13.3 using strategy S6. Hence, the size of the test suites can shrink to less than half.

For the *Extremes* model the number of generated test cases ranges from 356 using the strategy S3 to 60 using strategy S5 and only 28.3 on average when using strategy S6. Hence, by using a combination of random and fault-based test case generation one is able to shrink the size of the generated test suite to less than one tenth compared to a test suite created by a strategy that does neither of these optimizations.

In contrast to the significant reduction of the number of test cases, the total run-times for the test case generation process for each strategy stays almost constant. Table 6.3 shows the run-time of the purely fault-based strategies as well as the run-times of the combined strategy S6. The total generation time for one test suite using strategies S3–S6 ranges from more than two days (3928 minutes) to approximately four days (5863 minutes), depending on which model is used.

The table also presents the time spent by mutant-type. The first two rows show the time spent processing equivalent mutants: if the non-conformance between the original and the mutant cannot be shown within the given depth, processing just one mutant takes from 35 to 162 minutes. This is independent of the strategy and responsible for the largest portion of time spent. Hence, equivalent mutants are the main reason, why the total time spent on test case generation does not differ much among the presented strategies.

The next two rows show how much time is spent on processing mutants where test case generation fails due to a livelock. As already presented in Table 6.1 this only happens in the *Extremes* model. It can be seen that the livelocks in the mutants occur at the beginning of the exploration, therefore comparatively little time is spent on processing these mutants.

The number of mutants triggering a new test case depends on the strategy. In strategy S3, all mutants that are neither equivalent up to the search depth nor fail because of a livelock trigger the generation of new test cases. The average time needed for processing such a mutant highly depends on how sophisticated the killing approach is. For example, strategy S3 generates many short test cases, which lowers the average generation time. In the more sophisticated strategies S4–S6, this is often suppressed by the check whether an existing test case can already kill the mutant.

The last three rows show what happens if a mutant is killed by an existing test case: the average time to kill a mutant is comparatively low. For the *Extremes* model, which has a larger range for input values that need to be enumerated, it is below two minutes. For the *X_ error* model it is even faster taking less than half a minute in average.

Test case generation using a random walk is comparatively fast. The generation of each test suite containing three test cases took between two and three minutes.

Since strategy S6 (the combination between random and fault-based test case generation) has been repeated three times, Table 6.4 presents the associated data in detail.

Test Case Generation with S6	Extremes			X_error		
	Run1	Run2	Run3	Run1	Run2	Run3
Generated test cases (excl. random) [#]	21	28	36	13	14	13
Mutants triggering a new test case [#]	7	12	9	5	6	5
Mutants killed by existing test cases [#]	79	65	69	18	17	18

Table 6.4: Details on individual test case generation runs with strategy S6.

6.3.3 Test Case Execution

To measure the power of the generated test suites in the car alarm system case study an implementation has been created and faults have been injected using classical mutation testing. For the wheel loader case study this has been repeated.

Implementation As part of the master project of the author of this thesis a Java implementation of the ECU has been created. The implementation consists of approximately 500 lines of code and is based on the test model "EQC". Random test cases have been generated and the system has been implemented using a test-driven development method.

The design is similar to the implementation of the car alarm system. The implementation simulates an ECU, that receives its inputs by method calls and reacts by calling call-back methods, which are defined in two Java interfaces. One interface defines the methods of the ECU, which correspond to the ISOBUS messages of the real ECU. Additionally, there is a method called `tick()` defined in order to simulate time. Calling this method denotes, that one time unit has passed.

Test Driver The test driver is built analogously to the test driver of the Car Alarm System. It implements the environment interface to provide the call-back functions for the ECU.

Mutation Analysis The mutation tool μJava [MOK05] has been used again to create faulty versions of the implementation. All method-level mutation operators have been applied, which yielded 1511 mutated implementations. For manually inspecting, which of these mutated implementations are actual faulty and not just equivalent mutants, this number is by far too high. However, to be able to measure the quality of the test suites, which have been obtained using the model-based mutation approach, at least an approximation was needed.

In order to estimate, how many of the mutated implementations are equivalent mutants, the following procedure has been used: the test model "EQC", which has already served as basis for the implementation, has been used to generate 1000 test cases applying the random killing strategy (S7) of *Ulysses*. Mutated implementations, which survived this

Strategy	S3		S4		S5		S6		S7	
Model	X_err	Extr	X_err	Extr	X_err	Extr	X_err	Extr	X_err	Extr
Gen. test cases [#]	31	356	17	150	14	60	13.3	28.3	3	3
Mutation score [%]	36.01	72.7	36.01	72.7	35.85	71.27	36.99	71.61	28.84	61.14
Total mut. score [%]	81.07		81.07		79.65		79.87		65.47	

Table 6.5: Number of generated test cases and mutation scores for strategies S3–S7.

large test suite, have been considered equivalent. Note, that even though the random killing strategy (S7) is the fastest test case generation strategy, generating such a large test suite, took approximately three days on a single CPU. The execution of the resulting test suite on all mutated implementations consisted of a total of 174,903 test runs. Out of the 1511 mutated implementations, 317 mutants survived and have therefore been considered being equivalent to the original implementation. The remaining 1194 faulty implementations have been used to evaluate the power of the test suites generated with the *MoMuT* toolchain.

Results All generated test suites from the test models “Extremes” and “X_error” have been run against the 1194 faulty implementations. Table 6.5 shows the mutation score and relates them to the size of the generated test suites. Additionally the table shows the mutation score, that can be achieved by merging the two test suites of each strategy. The difference between the mutation score of the test suites are quite low. Having that in mind, strategy S6, is a good choice if the size of the test suite matters. Starting with a small number of random test cases and adding further test cases only, if existing test cases cannot kill a model mutant, is a good way to avoid creating short test cases. If on the other hand only mutation score matters, strategies S3 and S4 are slightly better.

Unfortunately, the killing strategies, in which the cut of the search tree is disabled, do not scale to complex test models, like the ones used in this case study.

It is also remarkable, that random test case generation is quite powerful. Having in mind, that a test suite containing 6 test cases, which can be generated in less than have an hour, can already kill about 65 % of the faulty implementations is quite impressive. However, it has to be noted, that the random killing strategy S7 still performs model-based testing using the model as oracle. Also, the model is used to guide the random test case generation, so that it creates meaningful test cases. Because of that, one should think of strategy S7 as random search through a test model rather than of purely random software testing.

Empirical data gained by this case study also supports the claim, that partial test models help for creating powerful test suites. In each strategy combining the test suites from both test models results in a higher mutation score than the better of the two distinct test suites, so each partial model contributes to the combined test suite.

7 Concluding Remarks

7.1 Summary

In this thesis the role of test models for a particular prototype toolchain performing model-based mutation testing has been investigated. This toolchain originated from a past research project as joint work between the Austrian Institute of Technology and the Institute for Software Technology. It takes UML models using class diagrams and state machines as input and generates abstract test cases. The UML models may include code written in OCL, as well as in a proprietary programming language called AGSL. The internal process of the toolchain includes a transformation of the test models into an intermediate language called "object-oriented action systems", which has a clearly defined formal semantics and can be mapped to a LTS (Labeled Transition System). As test selection criterion mutation testing is used. A set of faulty models is created and the conformance of their associated LTS to the LTS of the original model is checked. Test cases are generated, which are able to reveal non-conformance. As conformance relation Tretmans' *ioco* is used.

The main purpose of this thesis was to address the effects of possible modeling styles for the UML models, which are used as input for the toolchain.

One approach was the use of a series of partial models covering different functional aspects of the system-under-test in addition to or instead of one single full model covering the whole functionality. It has been argued, how a new iterative, test-driven development process could look like, which is based on model-based mutation testing using partial models.

In a first case study dealing with a car alarm system, it has been shown how a pre-existing model of a system-under-test can be decomposed into two partial models and how the toolchain can be used to generate a test suite in an iterative way. Furthermore three alternative UML modeling styles have been presented. Test cases have been generated and a mutation analysis using a pre-existing Java implementation has been conducted. Additionally, a comparison among different parameters controlling the test case extraction strategy of the test case generation process has been made. This gave new insight about the interdependency between the modeling style of the test model, the test case extraction strategy, and the results.

In a second case study dealing with the electronic control unit of a wheel loader, a pre-existing test model has been improved and complemented with a new additional partial model in order to be able to generate test suites in reasonable time and conduct a comparison of the different possible test case extraction strategies. A mutation analysis had been performed using a Java implementation, which had been created by the author of this thesis beforehand. The test case generation process within this case study has been a large empirical study, where alone the successful runs took over 40 CPU days.

7.2 Related Work on Partial Models

The idea to use partial models is not new, but rather state-of-the art in model-based testing. It has to be noted, that the term “partial model” in literature often is used for both horizontal and vertical modularity, while in this thesis it is only used to denote horizontal modularity.

In their textbook “Practical Model-Based Testing” Utting and Legeard [UL07] illustrate how to write test models for a drink vending machine using different notations. They emphasize the importance of the test model and show that partial models are possible and useful. This book also gives a very comprehensive overview over model-based testing practices in general.

Some early work in the field of partial models has been done by Larsen and Thomsen, who introduced the concept of “Modal Transition Systems” [LT88] in order to be able to explicitly define under-specified behavior, which refers to vertical modularity.

Other relevant work includes the work of Salay et al. [SFC12], who show how to define the information of uncertainty and partiality in a language-independent manner.

However, all of this work deals with partiality in design models, not in test models.

One example of research done in the field of partial models and compositionality for testing is the work of Grieskamp et al. from Microsoft Research [GKT06]. They use a formalism called “action machines”, which is similar to the concept of the “action systems”, which serve as intermediate language in the *MoMuT* toolchain. Their idea is to create so-called “scenario machines”, which can be used to slice the test model in order to gain a partial model. The “action machines” framework supports conformance checking using “alternating simulation”, which which was proposed by Alur et al. [Alu+98]. This relation is very similar to the *ioco* conformance relation used by the *MoMuT* toolchain. Veanes and Bjørner [VB10] even provide formal proofs that relate the refinement relation “alternating simulation” to *ioco*.

Another example for related work on partial models for model-based testing is the research of Robinson-Mallett et. al. [Rob+08]. Their approach is to generate partial models from test sequences in order to perform integration testing. The used formalism are timed automata.

There has also been work on improving the conformance relations, in order to improve the expressibility of under-specified behavior. This has been done by the group of Tretmans leading to the new conformance relation *uioco* [BRT04]. There the capabilities of *ioco* regarding under-specification have been extended. Also, for conducting vertical refinement steps a possibility to hide more abstract actions in favor of new more concrete actions has been introduced. This allows the modeler to replace an atomic action by a trace of actions. This technique is also referred to as *trace refinement* which is not to be confused with the *traces refinement* relation.

Other work in this field has been done by Gorrieri and Rensink, who suggested the idea of "action refinement" [GR01]. In their approach in each vertical level, the set of visible actions used can differ. In traditional *ioco* there is only limited support for this kind of refinement: in a later refinement step only additional input actions can be added, as explained in Section 4.1.

7.3 Discussion

In two case studies this thesis has demonstrated, how a system can be decomposed into partial models. Results show, that model-based mutation testing is applicable and is able to generate respectable test suites. Yet, it is not a push-button technology. A lot of parameters have to be chosen, including a wise selection of mutation operators leading to a meaningful fault-model or a search depth that is large enough to find the injected faults but small enough that equivalent model mutants do not delay the process exorbitantly. Another important parameter is the test case extraction strategy, that defines how many test cases are generated when non-conforming behavior can be identified. All this requires a thorough understanding of the used tools as well as of the test model. The test model itself is probably the most important factor for the power of model-based mutation testing.

In the car alarm system case study additionally to the pre-existing model three alternative modeling styles have been presented. The comparison among these modeling styles and different test case extraction strategies has shown very interesting interdependencies: for each modeling style there has been at least one parameter set which achieved 100 % mutation score on 38 faulty Java implementations. However a parameter set that worked perfectly for a test model using one particular modeling style, did not even finish in reasonable time for a test model of another modeling style. Some combination of parameters and modeling style led to test suites containing up to 700 test cases, while another combination led to a test suite containing only 11 test cases.

This thesis has shown both possibilities and limitations of a current prototype toolchain. In the case of the wheel loader demonstrator, before starting the work of this thesis, it seemed that test suites generated by a mutation-based approach would easily be outperformed by random test case generation. With a new partial model, developed within the thesis this has changed. For being so fast, random test case generation is still enviably good, but the test suites generated by a mutation-based approach have their advantages, too. Moreover, both techniques can be combined, leading to even better results.

From the wheel loader case study we also learned, how the toolchain scaled for large systems. Here partial models were not just an option, but instead they were needed in order to cope with the complexity.

7.4 Future Work

This thesis is only one step towards the applicability of model-based mutation testing. For one particular prototype toolchain it has been shown, how test models can be written and what impact a certain modeling style has on the results of a test case generation process. Even though the input to the toolchain is UML, which is a very common modeling language, creating good test models, is a very tricky task. It is not sufficient to be familiar with the input language itself, but also the knowledge of internal details of the toolchain is necessary in order to achieve satisfying results.

In order to distribute the toolchain to industry, a detailed set of instructions on modeling guidelines has to be developed. Insight gained from a work like this could be one source for doing so.

When building the models for the case studies of this thesis a lot of issues within some parts of the toolchain have been found. Some of them made it necessary to use little intuitive workarounds, other issues could hinder some approaches to succeed. In order to become applicable the prototype toolchain has to mature.

One particular issue is the lack of scalability to large models in terms of run-time and memory consumption. This is mostly due to the fact, that *Ulysses* uses an enumerative approach to build the state space, which inevitable leads to a “state-space explosion” unless all possible value ranges are restricted very rigorously. To overcome this problem, there is already ongoing work by Aichernig and Jöbstl [AJ12] where a symbolic approach is used instead of the enumerative approach.

Other ongoing joint work of the Institute for Software Technology and the Austrian Institute of Technology addresses the use of timed automata as formalism [ALN13]. This is a promising approach especially for designing test models, which have to represent complex time constraints.

Further use cases and demonstrators within current and future research projects will give insight on which approach is preferable in which domain and how the test models have to be designed.

Bibliography

[Abr+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. "Rodin: an open toolset for modelling and reasoning in Event-B." In: *International Journal on Software Tools for Technology Transfer* 12.6 (Nov. 2010), pp. 447–466. ISSN: 1433-2779. DOI: `10.1007/s10009-010-0145-y` (cit. on p. 4).

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* 1st. New York, NY, USA: Cambridge University Press, 2010. ISBN: 0521895561, 9780521895569 (cit. on pp. 4, 38).

[Abr96] Jean-Raymond Abrial. *The B-book: assigning programs to meanings.* New York, NY, USA: Cambridge University Press, 1996. ISBN: 0-521-49619-5 (cit. on pp. 4, 38).

[Aic+] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. "Killing strategies for model-based mutation testing." Submitted to Software Testing, Verification and Reliability (STVR) (cit. on p. 77).

[Aic+11a] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald 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 (cit. on pp. 7, 10, 31, 43, 70, 73, 91).

[Aic+11b] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. "UML in action: a two-layered interpretation for testing." In: *ACM SIGSOFT Software Engineering Notes* 36.1 (Jan. 2011), pp. 1–8. ISSN: 0163-5948. DOI: `10.1145/1921532.1921559` (cit. on pp. 16, 17).

[AJ12] Bernhard K. Aichernig and Elisabeth Jöbstl. "Towards Symbolic Model-Based Mutation Testing: Combining Reachability and Refinement Checking." In: *In Proceedings of the 7th Workshop on Model-Based Testing (MBT 2012).* Ed. by Alexander K. Petrenko and Holger Schlingloff. Vol. 80. EPTCS. 2012, pp. 88–102 (cit. on pp. 87, 100).

[ALN13] Bernhard Aichernig, Florian Lorber, and Dejan Ničković. *Model-based Mutation Testing with Timed Automata.* Tech. rep. Institute for Software Technology (IST), Graz University of Technology, 2013. URL: `https://online.tugraz.at/tug_online/voe_main2.getVollText?pDocumentNr=302132&pCurrPk=69694`(visited: 2013 May 6). (Cit. on pp. 73, 100).

[ALT12a] Bernhard K. Aichernig, Florian Lorber, and Stefan Tiran. "Integrating Model-Based Testing and Analysis Tools via Test Case Exchange." In: *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China.* 2012, pp. 119–126 (cit. on p. 39).

[ALT12b] Bernhard Aichernig, Florian Lorber, and Stefan Tiran. *Formal Test-Driven Development with Verified Test Cases.* Tech. rep. Institute for Software Technology (IST), Graz University of Technology, 2012. URL: `https://online.tugraz.at/tug_online/voe_main2.getVollText?pDocumentNr=275810&pCurrPk=67400`(visited: 2013 May 6). (Cit. on pp. 7, 39, 43, 44, 53, 69).

[Alu+98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. "Alternating Refinement Relations." In: *Proceedings of the 9th International Conference on Concurrency Theory.* CONCUR '98. London, UK, UK: Springer-Verlag, 1998, pp. 163–178. ISBN: 3-540-64896-8. URL: `http://dl.acm.org/citation.cfm?id=646733.759544` (cit. on p. 98).

[Bec02] Beck. *Test Driven Development: By Example.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530 (cit. on pp. 1, 39).

[BK83] Ralph-Johan Back and Reino Kurki-Suonio. "Decentralization of Process Nets with Centralized Control." In: *Proceeding PODC '83 Proceedings of the second annual ACM symposium on Principles of distributed computing.* ACM. 1983, pp. 131–142 (cit. on p. 16).

[BKS98] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. "An approach to object-orientation in action systems." In: *Mathematics of Program Construction.* Ed. by Johan Jeuring. Vol. 1422. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 68–95. ISBN: 978-3-540-64591-7. DOI: `10.1007/BFb0054286` (cit. on p. 17).

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series).* Addison-Wesley Professional, 2005. ISBN: 0321267974 (cit. on p. 3).

[BRT04] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. "Compositional Testing with ioco." In: *Formal Approaches to Software Testing.* Ed. by A. Petrenko and A. Ulrich. Vol. 2931. Lecture Notes in Computer Science. Berlin, Germany: Springer Verlag, 2004, pp. 86–100. URL: `http://doc.utwente.nl/66359/` (cit. on pp. 4, 98).

[BVW98] Ralph-Johan Back and Joakim Von Wright. *Refinement Calculus:* Graduate Texts in Computer Science. Springer Verlag, 1998. ISBN: 9780387984179. URL: `http://books.google.at/books?id=fRWQe23oB7kC` (cit. on p. 38).

[BWA10] Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. "Automated Conformance Verification of Hybrid Systems." In: *Proceedings of the 10th International Conference on Quality Software.* IEEE Computer Society, 2010, pp. 3–12 (cit. on p. 14).

[Cse+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. "VIATRA - visual automated transformations for formal verification and validation of UML models." In: *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*. 2002, pp. 267–270. DOI: `10.1109/ASE.2002.1115027` (cit. on p. 23).

[DF93] Jeremy Dick and Alain Faivre. "Automating the generation and sequencing of test cases from model-based specifications." In: *In FME93, International Conference on Industrial Strength Formal Methods*. Vol. 670. Lecture Notes in Computer Science. Springer, 1993, pp. 268–284 (cit. on p. 53).

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976 (cit. on p. 16).

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." In: *Computer* 11.4 (Apr. 1978), pp. 34–41. ISSN: 0018-9162. DOI: `10.1109/C-M.1978.218136` (cit. on p. 2).

[FSE10] FSEL. *FDR2 User Manual*. 2010. URL: `http://www.fsel.com/documentation/fdr2/fdr2manual.pdf`(visited: 2013 May 6). (Cit. on p. 39).

[Geo+95] Chris George, Anne E. Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan S. Pedersen. *The RAISE Development Method*. The BCS Practitioners Series. Prentice Hall Int., 1995. URL: `http://www2.imm.dtu.dk/pubdb/p.php?1862` (cit. on p. 38).

[GKT06] Wolfgang Grieskamp, Nicolas Kicillof, and Nikolai Tillmann. "Action Machines: a Framework for Encoding and Composing Partial Behaviors." In: *International Journal of Software Engineering and Knowledge Engineering* 16.5 (2006), pp. 705–726 (cit. on p. 98).

[GR01] Roberto Gorrieri and Arend Rensink. "Action Refinement." In: *Handbook of Process Algebra*. Ed. by J. A. Bergstra, A. Ponse, and S. A. Smolka. Elsevier, 2001. Chap. 16, pp. 1047–1147 (cit. on p. 99).

[Hoa78] C. A. R. Hoare. "Communicating sequential processes." In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: `10.1145/359576.359585` (cit. on p. 38).

[ISO97] ISO/IEC. *Information technology – Open Distributed Processing – Reference Model: Foundations*. ISO/IEC 10746-2. 1997. URL: `http://www.joaquin.net/ODP/Part2/9.html`(visited: 2013 May 6). (Cit. on p. 4).

[JH11] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing." In: *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 649–678. ISSN: 0098-5589. DOI: `10.1109/TSE.2010.62` (cit. on p. 2).

[Jon90] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990. ISBN: 0-13-880733-7 (cit. on pp. 4, 38).

[Kro10] Daniel Kroening. *MOGENTES deliverable D 3.2b - Modelling Languages (final version).* Tech. rep. 2010. URL: `https://www.mogentes.eu/public/deliverables/MOGENTES_3-13_1.0r_D3.2b_ModellingLanguages.pdf`(visited: 2013 May 6). (Cit. on pp. 20, 22).

[KSA09] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. "Mapping UML to Labeled Transition Systems for Test-Case Generation - A Translation via Object-Oriented Action Systems." In: *Formal Methods for Components and Objects (FMCO).* 2009, pp. 186–207 (cit. on pp. 5, 6, 20, 21, 23, 28).

[LT88] Kim G. Larsen and Bent Thomsen. "A modal process logic." In: *Logic in Computer Science, 1988. LICS '88., Proceedings of the Third Annual Symposium on.* -0 1988, pp. 203–210. DOI: `10.1109/LICS.1988.5119` (cit. on pp. 4, 98).

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. "MuJava: an automated class mutation system." In: *Software Testing, Verification and Reliability (STVR)* 15.2 (2005), pp. 97–133 (cit. on pp. 43, 95).

[Off92] A. Jefferson Offutt. "Investigations of the software testing coupling effect." In: *ACM Transactions on Software Engineering Methodology* 1.1 (Jan. 1992), pp. 5–20. ISSN: 1049-331X. DOI: `10.1145/125489.125473` (cit. on p. 2).

[Pol09] Balázs Polgár. *MOGENTES deliverable D 2.1 - Tool Integration Framework - Specification.* Tech. rep. 2009. URL: `https://www.mogentes.eu/public/deliverables/MOGENTES_2-02_1.4r_D2.1_Framework-Specification.pdf`(visited: 2013 May 6). (Cit. on p. 19).

[Rob+08] Christopher Robinson-Mallett, Robert M. Hierons, Jesse Poore, and Peter Liggesmeyer. "Using communication coverage criteria and partial model generation to assist software integration testing." In: *Software Quality Control* 16.2 (June 2008), pp. 185–211. ISSN: 0963-9314. DOI: `10.1007/s11219-007-9036-1` (cit. on p. 98).

[RS59] Michael O. Rabin and Dana Scott. "Finite automata and their decision problems." In: *IBM Journal of Research and Development* 3.2 (Apr. 1959), pp. 114–125. ISSN: 0018-8646. DOI: `10.1147/rd.32.0114` (cit. on p. 58).

[SFC12] Rick Salay, Michalis Famelis, and Marsha Chechik. "Language independent refinement using partial modeling." In: *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering.* FASE'12. Tallinn, Estonia: Springer-Verlag, 2012, pp. 224–239. ISBN: 978-3-642-28871-5. DOI: `10.1007/978-3-642-28872-2_16` (cit. on p. 98).

[Tir12] Stefan Tiran. *The Argos Manual.* Tech. rep. Institute for Software Technology (IST), Graz University of Technology, 2012. URL: `https://online.tugraz.at/tug_online/voe_main2.getVollText?pDocumentNr=275803&pCurrPk=67399`(visited: 2013 May 6). (Cit. on p. 28).

[Tre96] Jan Tretmans. "Test Generation with Inputs, Outputs and Repetitive Quiescence." In: *Software - Concepts and Tools* 17.3 (1996), pp. 103–120 (cit. on p. 10).

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 0123725011, 9780080466484 (cit. on pp. 1–3, 98).

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches." In: *Software Testing, Verification and Reliability (STVR)* 22.5 (Aug. 2012), pp. 297–312. ISSN: 0960-0833. DOI: `10.1002/stvr.456` (cit. on pp. 1, 9).

[VB10] Margus Veanes and Nikolaj Bjørner. "Alternating simulation and IOCO." In: *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems.* ICTSS'10. Natal, Brazil: Springer-Verlag, 2010, pp. 47–62. ISBN: 3-642-16572-9, 978-3-642-16572-6. URL: `http://dl.acm.org/citation.cfm?id=1928028.1928033` (cit. on p. 98).

[Wei08] Georg Weissenbacher. *MOGENTES deliverable D 3.1b - Fault Models (Final Version).* Tech. rep. 2008. URL: `https://www.mogentes.eu/public/deliverables/MOGENTES_3-09_1.0r_D3.1b_Fault_Models_Mutations.pdf`(visited: 2013 May 6). (Cit. on p. 27).