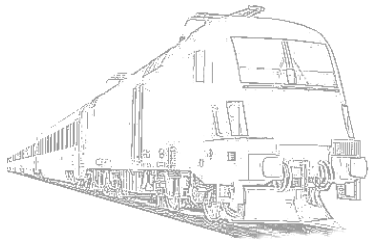
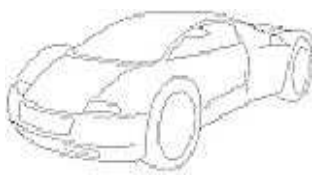
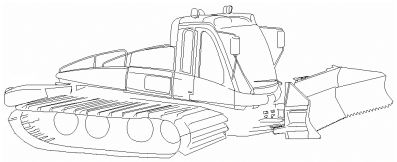
MOGENTES

Model-Based Generation of Test-Cases for Embedded Systems

Testing Theories and Coverage Criteria

Final Version

Project	MOGENTES		**Contract Number**		216679
Document Id	3-15_1.0r	**Date**	2010-06-30	**Deliverable**	3.4b
Contact Person	Bernhard Aichernig			**Organisation**	TUG
Phone	+43 316 873 5717			**E-Mail**	aichernig@ist.tugraz.at

Distribution Table

Name	Company	Department	No. of copies	Hardcopy/Softcopy
all MOGENTES cons. partners				

Change History

Version	Date	Reason for Change	PagesAffected
0.1w	2010-06-14	first draft (internal)	all
0.2w	2010-06-28	included input from BME	Sec. 4
0.3w	2010-06-30	internal final version	all
1.0r	2010-06-30	released	

Contents

List of Figures

List of Tables

1 Introduction

Both testing and verification techniques serve to increase our confidence in the 'correctness' of a particular piece of engineering. For these techniques to deliver reliable and sound results, a rigorous mathematical theory is needed. Put differently, the popular saying *"There is nothing so practical as a good theory"* (Kurt Lewin) perfectly applies in the given context. Since MOGENTES employs model-based test case generation, a solid theory is needed.

Scope and Purpose. This document presents the theory our model-based test case generation tools are based on. Because most of it has been published in papers or technical reports, we chose to make this deliverable two-parted: (1) The first part gives a concise overview of our theory and sets our publications in context. (2) The second part of this deliverable is formed by selected publications that explain theoretical concepts introduced in the first part in the very detail. In particular, the following list of topics is covered by our MOGENTES publications:

- The theory of combining action systems with Qualitative Reasoning (QR) models [6].
- A denotational semantics of the input-output conformance relation [3].
- The translation of UML models to (object-oriented) action systems [26].
- The conformance verification of qualitative action systems [19].
- Test case generation
  - from Labeled Transition Systems (LTS) [45, 25]
  - from Action Systems (AS) [4]
  - from Qualitative Reasoning (QR) models [5, 18, 17, 7]
  - for timed systems [39]
- Properties and the effectiveness of different test case selection strategies (based on action systems) [4].

Hence, in the first part we show how user supplied models are translated to suitable *intermediate representations*. We give the relevant *semantics* for these intermediate representations and show relations between them. In order to know when an implementation under test does (not) implement a given specification, we also present our chosen *conformance relation* and explicate design choices.

Before we start with theoretical aspects, however, we give conclusions that resulted from our survey of the state of the art [2] and have influenced our theory.

Partners Contribution. This document has been produced by TUG with contributions from AIT, BME, ETH, and other partners. The attached publications have been produced by TUG, AIT, BME, and ETH.

2 Conclusions From Survey

The survey done in Deliverable D1.2 [2] gave an overview of the wide body of existing work in the area of model-based test case generation (MBTCG). It also contained recent trends and conclusions drawn from other projects concerned with MBTCG. Conclusions drawn from the survey largely fall into two categories: (A) general observations and (B) observations that influenced MOGENTES. In the remainder of this section we briefly go over these, starting with general ones. Based on these observations, we finally identify areas where research that is done in MOGENTES goes beyond the state of the art.

2.1 Observations

General Observations. On the general level, observations from the survey are among the following. Probably the most intriguing one is that MBTCG seems on the way to gain wider acceptance by industry: Commercial tools like All4Tec's MaTeLo, Conformiq's Qtronic, Smarttesting's Test Designer (former Leirios Test Generator), Siemens' TDE/UML – among others – indicate the feasibility of MBTCG within an industrial context. Besides, it is also remarkable how much research projects connected to testing (and verification) are pursued by "big players", e.g., Microsoft. In summing up, it is justified to say that testing and verification techniques are under the most sought-after techniques today and that the importance still is increasing.

While industry is becoming more and more interested in MBTCG, the survey also shows open issues that await a solution. Most notable is the inherent complexity of automated test case generation from (industrial-sized) models: Tools like TorX that do on-the-fly random test case generation have less complexity issues to battle than e.g., TGV which does scenario-based offline test case generation (cf. AGEDIS project). Also, there is the indication that the generation of necessary models and test purposes (test scenarios) is a task that needs specialised engineers: There is a learning curve involved that cannot be totally avoided.

On a related but different matter, the quality of automatically generated test cases was not thoroughly discussed in the survey: Most of the cited work was concerned with coverage-based quality metrics. While coverage-based quality indicators are indeed important building blocks, a test suite of millions of automatically generated tests does not help when there is no time to run it. Hence optimising the quality of automatically generated test cases remains a valuable research topic for MOGENTES.

As last observation, we point out that there is few work on fault-based testing. Because MOGENTES aims at this area of testing, there are lots of opportunities to advance the state of the art.

Project Relevant Observations. Within Section 5, the survey features discussions of industry-based case studies. Most influential of these were the points summarised under the 'lessons learnt' paragraph from the AGEDIS project. Since the use of UML/OCL as front-end language was one of the lessons learnt of the AGEDIS project and – at the same time – also of great importance to the MOGENTES industry partners, a subset of *UML/OCL* (besides Simulink) now is the agreed modelling language adopted in MOGENTES.

While UML/OCL seems fine as a front end language, it was not considered as input language to test case generation tools, as it was thought to be too high-level and semantically imprecise. Instead, the idea of *model programs*, like in SpecExplorer, was adopted. This decision was also backed by the fact that Simulink-based specifications can be 'naturally' mapped to C programs and that there are similar ways to extract model programs from UML/OCL diagrams. Hence, model programs turned out to be a common concept able to support both Simulink and UML/OCL models. However, instead of using C code as intermediate language, the consortium members agreed on an intermediate language based on *action system* semantics (and subsets thereof). The decision is backed by the body of existing work (e.g., refinement theory, hybrid action systems, object-oriented action systems that build a bridge to UML models), the versatileness, and the general fitness of the formalism for the industry partner's demonstrators.

The survey also lists work in the area of *hybrid action systems*. Within MOGENTES, we extend the proposed hybrid action system theory to that of *qualitative action systems*. Qualitative action systems join techniques from artificial intelligence, namely qualitative reasoning, with hybrid action systems. The resulting qualitative action systems are a natural fit to describe hybrid systems having very abstract requirements.

Other influential notions within the survey were *test purposes* that can be thought of as some sort of test case scenarios and the works on *mutation testing* which MOGENTES will be contributing to.

2.2 Progress Beyond State of the Art

Based on these observations, we see several directions in which project activities aim at progressing beyond the state of the art described in the survey.

Action Systems. The presented action-system-based test case generation work flow advances the state of the art in the following areas:

- Qualitative action systems are a new formalism that joins techniques developed in the formal methods and artificial intelligence communities. The strength of qualitative action systems lies in the ability to model hybrid systems with highly abstract continuous behavior that could not (or with great difficulties) be modeled in other formalisms. As an example, continuous behavior specified in requirements documents often lacks the precise information needed to map it directly to ordinary differential equations. Anyway, one might just not be interested in the very details of the continuous behavior.
- Within MOGENTES, an extension to the object-oriented action systems published in [16] is used. In particular, we add the notion of prioritized composition [42], and more sophisticated data types such as tuples and lists. To the authors' knowledge there has been no previous attempt to use these types of action systems for test case generation. Also, the mapping of UML to object-oriented action systems [26] as done in MOGENTES is considered new work. Note that a mapping of action systems to UML has been published in [46], but there exists no previous work that maps standard UML elements to action systems. Furthermore, we extend (object-oriented) action systems by labeling each action with a name. The execution semantics of such an action system is then defined as a Labeled Transition System (LTS) which enables us to apply standard model-based testing techniques.
- Lastly, test case generation by (specification) mutation in connection with the ioco-family of conformance relations is rarely used. This together with models of industrial size advances the state of the art.

Fault Injection. A problem related to fault injection is the selection of input stimuli, which will exercise fault handling mechanisms in the target system. In MOGENTES, support for effective test case generation for embedded systems is developed. Thus, an interesting idea for a new research activity within the project is to use a MOGENTES test case generation tool to feed the MODIFI tool with test case suites that, e.g., will exercise all execution paths in a model. The same test cases should be used for the golden run (fault-free run) and for all subsequent experiments, where different faults are injected into the executed model. The experiments will be compared with the golden run to measure the number of detected errors (DE) and the number of wrong outputs (WO) where the ratio DE/(DE+WO) is denoted as the error detection coverage (EDC). This way, faults can be activated during execution of all paths which may reveal robustness problems (e.g., result in more WOs) not found by using manually selected stimuli. Another advantage is that error detection and recovery mechanisms inserted in any branch of the model can be activated and evaluated. Thus, the confidence of the measured EDC will be increased and the problem of selecting stimuli will partly be solved.

Minimal Cut Sets. Generation of minimal cut sets (MCS) based on formal verification of safety requirements as performed in MOGENTES leads to new questions and challenges, providing input for new research directions.

One issue is the speed and capacity of the formal verification process in itself. When systems scale in size and the number of introduced faults grows, will formal verification be able to provide adequate response times in MCS generation? In practise, this is not a huge bottleneck if using state-of-the-art formal verification technology and reducing the number of faults to one. Allowing more faults cause and large systems require longer time in MCS generation. Tuning the formal verification technology to cope with this challenge is one area for future research.

Another main issue is related to how an overall development process should make use of MCS generation, or test cases based on MCS. It is usually straight forward to identify that certain isolated faults should not be allowed to cause safety hazards, and hence the question of how to use the results of the MCS generation is straight forward in such cases. For instance, one isolated hardware fault in a relay-based railway interlocking system should not jeopardize safety. However, it is less clear how an overall development process in general should incorporate the use of formal verification-based MCS generation for computerized railway signalling systems. For instance, what potential faults in a computerized railway control system are worthwhile to include in a formal verification-based MCS generation, taking into account that hardware platforms used for such safety-critical platforms usually make use of redundant systems in order to detect various types of faults (and enter a safe state in such cases). Future research aiming to define an overall development process that incorporates formal verification-based MCS generation is clearly needed.

Test Case Generation for Simulink. During the model analysis, tools such as the Simulink Design Verifier rely on approximations of floating point arithmetic by means of infinite-precision rational numbers (according to

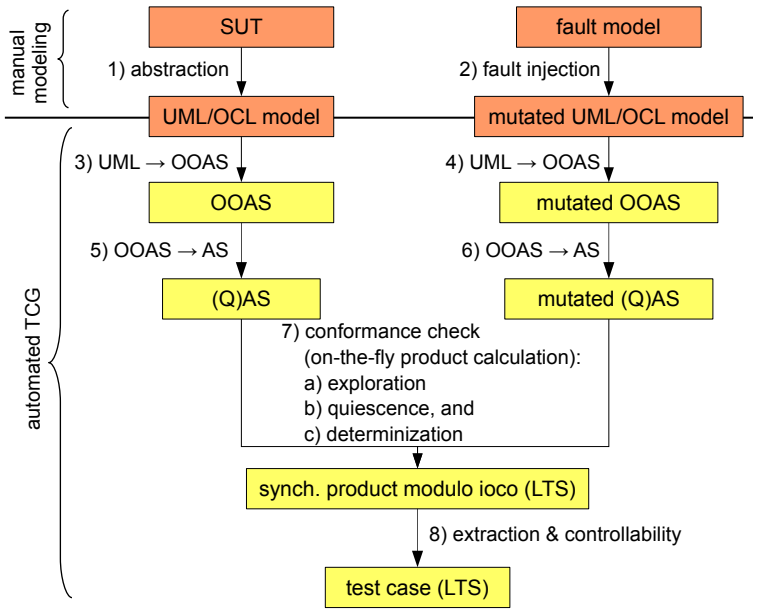


Figure 1: Overview of the test case generation process based on action systems.

the Simulink Design Verifier User's Guide). The COVER tool developed in the MOGENTES project uses an bit-level-accurate representation of floating point arithmetic and is therefore able to analyse the exact behaviour of the model rather than an approximation. Furthermore, our bit-level-accurate model checking technique enables the use of mutations such as single bit-flips in data values.

Similar to state-of-the-art test case generation tools such as the Simulink Design Verifier, our tool COVER is able to generate test cases that satisfy model coverage objectives such as structural coverage. An improvement beyond the state of the art is the ability of COVER to generate test cases that "cover" mutations (i.e., detect the presence of a predefined set of mutations of the model). We devised a lattice-based technique to efficiently generate test suites that achieve mutation coverage.

3 Test Case Generation on the Basis of Action Systems

This section presents the basic concepts underlying test case generation (TCG) on the basis of action systems [9, 10]. More precisely, we present the test case generation work flow depicted in Figure 1. Starting from an UML/OCL model of the system under test (SUT), fault models are used to obtain a mutated version of the system. Both UML models, the mutated as well as the original one, are then mapped via object-oriented action systems (OOAS) to action systems (AS). Afterwards, we perform a conformance check by exploring the action systems and thereby obtain an LTS representing the synchronous product modulo ioco, from which a controllable test case will be derived. Note that this approach allows the generation of test cases from qualitative action systems (QAS), but that there is no modelling support of hybrid systems on the UML level. Notice further that action systems are our chosen intermediate representation and that different partners use subsets of this formalism.

3.1 A UML Model

The content of this subsection is covered in the technical annex [26]. Also, D3.2b contains a discussion of supported UML elements, while D3.1b covers mutation operators.

Because UML is the predominant standard for modelling software within industry today and due to the outcome of the survey, the MOGENTES partners quickly came to the conclusion that the front-end language of choice should be based on UML. At the same time, however, it was also decided that test case generation had to be

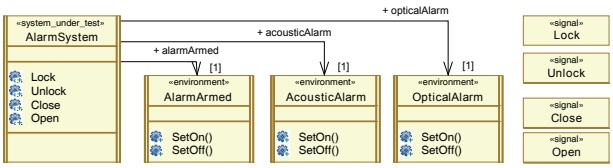


Figure 2: Testing interface specified in UML.

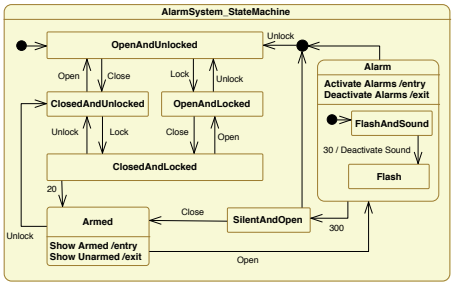


Figure 3: UML state machine of the car alarm system.

"Types"	Class Enums Signal Bool Int	Classes	Active/Passive Associations "Simple" Inheritance Member Fields Methods Def. + Body Signal Reception	OCL	and, or, not, implies $=, <, >, <=, >=$ union, intersect select, collect exists, forall, oclIsInState Literals (Numbers, Bool)
State Machines	Substate Machine Orthogonal Regions (Final-, Initial-, Pseudo-) State Entry, Exit Action Transitions with Effects Trigger with Change/Signal/Call/Time Events Constraints (OCL) Junctions, Choice				

Table 1: Supported UML Elements.

based on some simpler intermediate representation. Due to reasons already discussed in the previous section, action systems were chosen to be the theoretical basis of this intermediate language.

Figures 2 and 3 show excerpts of the UML/OCL model of Ford's car alarm system (CAS) demonstrator. Modelling was done by AIT with an open source tool called Papyrus. (See Table 1 for a list of supported UML elements.) Figure 2 shows the UML diagram specifying the test interface, while Figure 3 shows the state machine underlying the system under test. The main purpose of this example is to test and validate the test case generation work flow on a basic level.

Testing Interface The UML model of the car alarm system comprises four classes and four signals, as shown in Figure 2. The class *AlarmSystem* is marked as system under test (SUT) and may receive any of the *Lock*, *Unlock*, *Close*, or *Open* signals. At the same time, the SUT calls methods of the classes *AlarmArmed*, *AcousticAlarm*, and *OpticalAlarm* – all of them marked as being part of the environment.

Notice that the context diagram in Figure 2 specifies the observations we can make (all calls to methods being part of the environment) and the stimuli the system under test can take (all signals). Hence, this diagram specifies our *testing interface*.

State Machine Figure 3 shows the CAS state machine diagram. The state machine is small but non-trivial: it involves hierarchical states, time triggers, and entry-exit actions with methods defined in class diagrams.

Although we strive for a UML-conform semantics, our chosen semantics differs slightly from the UML standard: In order to support partial test models, the state machine only accepts events that trigger a transition: in the CAS model the state machine will only accept *Close* and *Lock* events when being in state *OpenAndUnlocked*. We also use a simplified notion of time that restricts the model's behavior when multiple time-triggered transitions are used simultaneously.

Mutating the UML Model As we want to create test cases that cover particular fault models, we need to deliberately introduce 'bugs' in the specification model. In order to do that, we rely on different mutation operators. As an example, one mutation operator sets guards of transitions to false, while other ones remove entry actions, signal triggers, or change signal events.

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

3.2 Action Systems as Intermediate Models

The content of this subsection is covered in the technical annex [26].

Conventional action systems [9, 10] lack support for a set of features used within the UML model. Most prominently, action systems lack class and object support. Also, a way of prioritizing actions needs to be added to the formalism. Both shortcomings have been independently addressed by researchers in the past: Bonsangue et al. [16] introduce an object-oriented extension of action systems, and Sekerinski et al. [42] add a prioritizing composition operator. That said, our work is the first to combine object-oriented action systems (with custom extensions), prioritized composition, complex data types, and an event trace semantics. In the following, we give an overview of action systems including the object-oriented and qualitative extensions used in MOGENTES. We also present a trace semantics for action systems.

3.2.1 "Conventional" Action Systems

We represent an action system AS comprising m functions, a named actions, d non-deterministically composed anonymous actions, and a set F_I of imported functions syntactically as follows:

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

Notice that functions have a name, a body and may return a value. Named actions are similar to functions but may not have a return value. The box operator (" $\square$ ") stands for non-deterministic, demonic choice. Demonic choice of actions means that when an aborting action is enabled, this action is chosen.

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

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

Within this representation, V is a vector of variables of types T, initialized with values I, and all A_i $(1 \leq i \leq d)$ have been subsumed under action A: see Table 2 for a list of actions. Notice that besides non-deterministic composition of actions, also sequential and prioritizing compositions are allowed. Finally, after inlining all imported functions $\in F_I$, Z denotes the set of imported variables of the environment that was accessed by the imported functions.

After eliminating all function calls, the action system consists of basic actions only (see Table 2) and all actions are part of the **do od**-block, also known as Dijkstra's guarded iteration statement [20]. The guarded iteration

Action	Notation	$wp(\textit{Action}, q)$
Sequential Composition	$A_1; \ldots; A_n$	$wp(A_1, wp(\ldots, wp(A_n, q)))$
Nondeterministic Composition	$A_1 \,\square\, A_2$	$wp(A_1, q) \wedge wp(A_2, q)$
Prioritizing Composition	$A_1 // A_2$	$wp(A_1, q) \wedge$ $(\neg g.A_1 \Rightarrow wp(A_2, q))$
Guarded Command	requires p: A_1 end	$p \Rightarrow wp(A_1, q)$
Multiple Assignment	$y := e$	$q[y := e]$
Nondeterministic Assignment	$z := z'$ with Q	$(\forall z' \in Q.z \cdot q[z := z'])$
Local Variables	*var* $x_1 : T_1; \ldots; x_n : T_n : S$	$\forall x_1 \ldots x_n : wp(S, q)$
Skip	**skip**	q
Abort	**abort**	*false*

Table 2: Semantics of Basic Actions.

statement can be thought of as being a loop that selects one enabled action A_i for execution in each iteration. In case there is no action enabled, execution of the action system ceases as execution of the loop terminates.

To determine whether an action A is enabled, we compute the enabledness guard: As we do not want an action A to be enabled for states in which it is guaranteed to establish any postcondition, i.e., behave miraculously, the enabledness guard is defined as $g.A =_{df} \neg wp(A, false)$, where $wp : Action \times (State \mapsto Bool) \mapsto (State \mapsto Bool)$ is the weakest precondition predicate transformer. For example, the precondition of a guarded command is given by

$$wp(\text{requires } \textit{guard} : \textit{A} \text{ end}, q) \equiv \textit{guard} \Rightarrow wp(\textit{A}, q)$$

with "⇒" denoting logical implication. Table 2 lists the weakest preconditions of all actions for any given predicate q. In Table 2, A_1, A_2 denote actions, y stands for a list of variables, z, z' are variables, e is a list of expressions, p is a predicate over the state, $g.A_1$ is the enabledness guard of action A_1, and Q is a predicate over z, z' (and the state). The nondeterministic assignment assigns to variables z a value of z' for which Q holds. The statement aborts if this is not possible [10]. Notice that an action will terminate if the *termination guard* $t.A = wp(A, true)$ holds.

3.2.2 Extension A: Object Orientation

We use the work of Bonsangue et al. [16] as the basis for object-oriented action systems (OOAS): in particular we share the transformation step from object-oriented action systems to action systems. We differ in the notion of named actions and procedures and we add the ability to prioritize objects of a particular class with respect to objects of another class. Within our methodology, we use a very simple form of inheritance: A class C^2 is a valid subclass of C^1 if and only if the (syntactic) superposition (cf. [11]) refinement holds between the classes. Roughly speaking, this means that C^2 may introduce additional variables and actions. However, none of the additional actions may have any effect on the variables of C^1, it must be guaranteed that when only considering the new actions and the initial state the system terminates, and the exit condition of C^2 must imply the exit condition of C^1. The subclass C^2 may override (refine) actions of C^1 in a way that the guard is strengthened and values to the additional variables are assigned.

We restrict an object-oriented action system to a finite set of classes $\mathcal{C} =_{df} \{C^1, \ldots, C^k\}$ and a finite set of objects. Practically, this means that we allow object instantiation only during state-variable initialization, which permits us a rather easy check of finiteness.

We assume that all objects of one class have the same priority. Between objects of different classes, however, we allow ordering with the help of the prioritizing composition operator: we introduce a so-called system assembling block (*SAB*). The *SAB*, which is an extension to the work of [16], specifies the ordering of priorities between objects of different classes. We rely extensively on this feature in order to model, e.g., event broadcasting. The syntax of the system assembling block is defined by the following grammar:

$$\textit{SAB} ::= C_n \; ((\,\square \mid //\,) \; \textit{SAB})?$$

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

The semantics of object-oriented action systems is given by a mapping to action systems which is based on the work presented in [16]. The main idea of the mapping is to create one action system per object and join all action systems as specified in the system assembling block.

3.2.3 Extension B: Qualitative Actions

The content of this subsection is covered in the technical annex [6] and [7].

In order to model hybrid systems on an abstract level we introduce Qualitative Action Systems [6] as an extension to conservative action systems[1]. Here, the continuous behavior of hybrid systems is represented by so-called Qualitative Differential Equations (QDEs). QDEs are an abstract representation of Ordinary Differential Equations (ODEs). Qualitative Reasoning (QR), a technique from Artificial Intelligence, is used to infer behavior from a set of QDEs and a given initial state. This is the analog to the initial value problem of ODEs. Inspired by the work in [41] we define our qualitative actions via weakest precondition semantics (cf. [6]). This results in all continuous evolutions to be abstracted to their pre- and post states. In other words, the continuous evolution itself is not observable. Instead, we associate the post-state of a qualitative evolution with an (observable) event.

In [7] we elaborated the testing of continuous systems by using qualitative models. In contrast to qualitative actions where only the pre- and post-states of continuous evolutions are considered, this work aimed at testing such evolutions.

3.2.4 Trace Semantics of Action Systems

The content of this subsection is covered in the technical annex [19] and [26].

By associating names to all actions we give our action systems a labeled transition system (LTS) semantics. The LTS of an action systems is obtained by exploring all possible execution sequences starting from a given initial state. Since qualitative actions are also mapped to events, this abstraction also works for hybrid systems. Although the abstraction of continuous evolutions to discrete events is quite strong the approach is well suited to restrict a controller's environment and to detect faults in such continuous environments.

3.3 Mapping UML to OOAS

The content of this subsection is covered in the technical annex [26].

In the transformation, we strive for following the UML v2.2 standard. Nonetheless, we have made some design choices, aside from the selected elements: object instantiation is limited to the initialization phase while destruction of objects is not used at all. This fits well into the current practice in embedded systems design, where a constant, limited and predictable memory footprint is wanted. This also avoids some of the semantic variation points on deletion/creation in context of composition and aggregation relations between classes.

Classes, member fields and method definitions map easily to the respective counterparts in the OOAS as described in the previous section. Mapping of inheritance is also straight forward, provided the subclass is a valid superposition refinement of the superclass. Behavioral aspects are mainly expressed with state machines in the selected UML subset. While there is a similarity between state machine transitions and guarded actions, some of the features of UML state machines need some more thought on how to implement them in OOAS. Briefly speaking, we map concurrency to standard conformant non-deterministic choice, treat event processing as in-order and loss-less, and support time-triggered transitions via timer queues.

Notice that some of our design decisions give our models a behavior that deviates from the UML standard: This is the case for time-triggered transitions and for the input-enabledness of the model. See [26] for details.

[1] Qualitative actions can also be added to object-oriented action systems.

3.4 Input Output Conformance (IOCO)

A denotational semantics of IOCO is given in the technical annex [3].

The conformance relation ioco has been developed by Tretmans for testing the behaviour of communicating systems [43]. Focusing on communication events, it is defined over models with a labelled transition system (LTS) semantics, the labels representing the communication events.

Informally, ioco conformance is given, if for all event traces possible in the specification model, the implementation does not produce output that is not allowed by the specification.

Like the refinement of pre-postcondition specifications, it supports incomplete (partial) specification models. This is realised by splitting the events into input and output events, i.e., controllable and observable events. Hence, an SUT may react to unspecified input events in an arbitrary way, like programs outside their specification precondition. This is an important feature for industrial applications, where it is unrealistic to model the complete behaviour of a system under test.

A further advantage of ioco is that it allows quite general specification models. They may be non-deterministic or have arbitrary interleavings of input and outputs in the models. For example, a specification may prescribe that in a certain state the system may accept input but alternatively may also issue output.

Important in practical software testing is the detection of absence of (output) reaction. Therefore, ioco adds an additional observation called *quiescence* to its alphabet of output events. This observation of quiescence can be easily implemented via timeout watchdogs reporting the absence of response from the SUT in a given time limit. Hence, with ioco one can test global real-time response times.

For this reasons, ioco has been the choice in our work on testing communicating systems. However, ioco is also based on assumptions of which some are only informally stated. Therefore, in [3] we have reexamined ioco and reformulated it in a denotational, predicative semantics making the assumptions explicit in form of healthiness conditions over the theory of reactive processes. This work also lifts the problem of ioco checking to the symbolic level, facilitating the application of modern SAT Modulo Theory (SMT) solvers, like Microsoft's Z3 or SRI's Yices tools.

For a discussion on possible alternatives to the ioco conformance relation we refer to our state-of-the-art survey (Deliverable D1.2 [2]).

3.5 Fault-Based Test Case Generation

Test case generation is covered in the technical annex [19], [5], [4], and in D4.3.

In the following, we only sketch the most relevant ideas for test case generation, since it is covered in detail in Deliverable 4.3.

Our test case generation is based on changing (mutating) a given specification according to some fault model, afterwards comparing the mutated version of the specification with the original one, and – in case a difference is found – using this difference as a test case. In general there are different notions of equivalence, or conformance. For our tools, we use the input-output conformance (ioco).

For mutation-based test case generation from (qualitative) action system we determine the input-output conformance (ioco) between an original and a mutated specification. By interpreting action systems via their LTS semantics we compute the synchronous product modulo ioco between two given action systems. The result of the ioco check is again an LTS which expresses the distinguishing behavior between the two LTSs. Non-conformance is indicated via *fail* states which serve as goal states for subsequent test case selection. The definition of the trace semantics of (qualitative) action systems as well as the ioco check are described in [19].

The work in [5] deals with the generation of mutation-based test cases for hybrid systems. A main topic of the paper is the discussion about controllability of generated test cases. During determinization of the LTS of an action system the situation can arise that in some states certain events are not always enabled. Their enabledness depends on internal state information which is lost during the calculation of the visible system behavior. We propose a rule which resolves this controllability conflicts while preserving the ioco relation. Furthermore, the paper presents first test cases generated from an qualitative action system.

	A1	A2	A3	A4	A5	A6	A7	A8
Max. Depth	10	14	23	23	23	150 (19)	150	30
Gen. TCs [#]	16 210	302	504	129	63	11	3	9
Unique [#]	3 469	110	269	123	59	11	3	9
Gen. Time [min]	188	91	23	70	23	10	0.25	-

Table 3: Number of generated test cases

In [4], we compare eight different test case generation approaches with the help of the car alarm system (CAS) demonstrator of Ford. Six of the investigated approaches are fault-based, i.e., relying on the conformance checking techniques developed within the MOGENTES project. The other two approaches are random testing and test purpose-based testing (TGV [24]). They are included for a better comparison with existing methodologies.

All of our six fault-based approaches start from the result of our conformance check: the synchronous product modulo ioco, which will be named *test graph* in the following. As already mentioned, this LTS contains so-called fail states indicating differences between the original and the mutated specification. Note that all approaches differ in their coverage of the test graph. In the following, we give a short description of our approaches (named A1 to A6). Table 3 gives an overview of their outcome.

Approach A1 transforms the test graph into a tree with a specified maximum depth (10 for the CAS) and a given maximum number of visits per state per trace (2 for the CAS). Afterwards, test cases are generated so that all paths that lead to a fail state are covered. As can be seen in Table 3, this strategy produces about 3500 unique test cases.

Approach A2 works directly on the product graph and extracts just one arbitrary test per fail state. Note that one injected fault may still involve more than one fail state in the test graph. This approach generates 110 unique test cases. Since we allowed for tests with a maximum length of 14 in this approach, the results are not directly comparable to those of A1.

Approach A3 is similar to A2 except that the depth is theoretically unbounded. Due to optimizations of the test case generator, the generation time is not comparable with approaches A1 and A2 (but with A4 to A7).

Approach A4 builds on A3 but avoids creating duplicate test cases. So while the total number of generated test cases decreases by 375, the time used to generate them increases by 47 minutes. A4 checks whether an existing test case already covers a fail state in the test graph before creating a new test case. If a particular fail state (there may be several per mutated specification) is not covered, a new test case is emitted.

Approach A5 also avoids creating duplicate test cases. However, A5 further tries to minimize the size of the generated test suite. Before creating test cases for a mutated specification, A5 first checks whether any of the previously created test cases is able to kill the mutant. This check is done without calculating the full *ioco* product between the specification and the mutant, which is the first difference to A4. Secondly, A4 will only skip test case generation for specific, covered fail states in the test graph, while A5 does never generate any additional test case for a killed mutant. Due to this minimization, approach A5 is sensitive to the ordering of mutants.

Approach A6 is like A5, but instead of starting with an empty test suite, A6 uses one randomly generated test case to start with. In Table 3, the maximum depth not put in brackets is the depth of the random test case while the figure in brackets is the maximum depth of the additionally generated tests to cover all faulty specifications.

Approach A7 denotes the random test case generation strategy and **Approach A8** is based on manually designed test purposes.

Limitations Our mutation testing approach relies on checking the input-output conformance of two models, which is a kind of equivalence check. It is well-known that equivalence checking of two algorithms is undecidable in general. We counter this problem with four approaches:

- *Bounded Models* allow us to completely unfold the search space during conformance checking. If the search space is still to large we stop exploration at a defined search depth, similar to bounded model checking. Hence, we check for equivalence up to a certain point.

- *Partial Models* are used to express a subset of a system's behavior, i.e., test scenarios, which limits the search space.
- *Abstraction* provides a further step of state space reduction. Besides identifying equivalence classes we apply qualitative abstraction to continuous behavior.
- *Random Search* enables us to generate test cases for easy to spot mutations in less time.

Since we are using ioco, the treatment of time is reduced to timeout events. We cannot distinguish between transitions with different timing constraints, i.e., we cannot compute the interleaving of timed traces. Where complex time is an issue, UML state machines are translated into timed automata as explained in the next section.

4 Test Case Generation on the Basis of UML State Machines with Time Extensions

For applications with stronger real-time needs, the previous approach appears less appropriate. Instead, a somewhat different approach is investigated in this section.

Introducing timing-related aspects into state-oriented models is a challenge in case of MOGENTES demonstrators characterised by real-time behaviour. This section outlines our proposal for (i) the *representation* of *time-related features* in state charts (called UML State Machines with Time Extensions, UML SMTE in short), (ii) assigning unambiguous *formal semantics* for UML SMTE models and (iii) a straightforward method for the automatic generation of *test cases* on the basis of UML SMTE models.

4.1 Modelling Time in State-Based Systems

UML provides its *state machine concept* and *state chart diagrams* for modelling event driven finite state transition systems. States may be organised into a *refinement hierarchy* of any level also supporting concurrent decomposition, state machine embedding etc. A rich variety of *pseudo-states* is offered for the modeller to indicate static or dynamic choice points, forking into and joining from concurrent regions, remembering previous sub-configuration, termination, etc. Compound *transition structures* are allowed to be built up of any number of basic transitions and pseudo-state vertices. *Activities* may be assigned to entering and leaving states and firing transitions. Transitions are triggered by reception of an (optional) event and may also be guarded by side effect-free Boolean predicates.

State machines operate according to the *run-to-completion (RTC) paradigm*, i.e., the reception of an event may trigger any number of transitions, and firing of a maximal set of non-conflicting firable transitions may take the state machine to a configuration where any number of transitions without triggers become enabled; an RTC step is closed if there are no more transitions that can be fired. An RTC step is considered to be performed instantaneously, i.e., there is no "duration" of an RTC step thus there is no duration of performing any activities or evaluating guards.

In contrary to the rich modelling toolkit for building state hierarchies and compound transitions structures, UML provides only relatively rudimentary means for modelling time-related aspects in state charts as outlined below.

- As state machines expose event driven behaviour, the elapse of time is primarily mapped to *time events* (metaclass TimeEvent). According to the standard, a time event specifies an *instant in time* by an expression. The expression might be *absolute* or it might be *relative* to some other point in time. Relative time events must always be used in the context of a trigger and the starting point is the time at which the trigger becomes active. With respect to the visual representation of time events the standard states that: A relative time trigger is specified with the keyword 'after' followed by an expression that evaluates to a time value, such as "after (5 seconds)." An absolute time trigger is specified with the keyword 'at' followed by an expression that evaluates to a time value, such as "Jan. 1, 2000, Noon".
- In the *context of state machines* the time event concept (i.e., the TimeEvent metaclass) is extended by a special interpretation of its "relative" nature: "If the deadline expression is relative and no explicit starting

time is defined, then it is relative to the time of *entry into the source state of the transition* triggered by the event. In that case, the time event occurrence is generated only if the state machine is still in that state when the deadline expires" (indicated by prefixing the time expression with the 'after' keyword in triggers of transitions as mentioned above).

The base UML standard does not discuss either the interpretation of *time* as a measurable entity or the modelling of *clocks* but leaves these questions open as semantic variation points. Since modelling and measuring time may be a major aspect in case of real time systems where physical clocks have to be represented, OMG proposed its UML Profile for Modelling and Analysis of Real-Time Embedded Systems (MARTE).

Our contribution to UML based time modeling and formal design methods is the clear definition of time events' applicability in state machine models and automatic transformation of such state machines to timed automata (input formalism of the UPPAAL tool environment [29, 28]). Time events can be used in state machines as follows:

- Time events can be assigned to transitions as trigger events. The time event must be set to *relative*. The "when" attribute of the time event should be a time expression specified by a string literal. Currently the value of the string is interpreted as an integer used for comparisons with UPPAAL clocks.
- The semantics of time events used as triggers is quite straightforward. Let t be a transition triggered by a time event whose source state is s; the time expression in the event should indicate T in its "when" attribute. After entering s an alarm clock is started that goes off after T time units and the corresponding time event is inserted into the event queue. Upon leaving s before the alarm the clock is shut down.

Note that in this interpretation time events are just ordinary events that are processed in FIFO order thus it may happen that at the actual processing of the time event the source state is not active any more. In these cases care should be taken if the same time event is re-used at multiple places of the state machine.

4.2 A Formal Semantics for UML SMTE

More details on the semantics for UML SMTE can be found in the technical annex [39].

For model checking and automatic test generation we definitely need an unambiguous formal semantics for (i) data modeling, (ii) state machines and (iii) specification of activities' internals. Our efforts concerning the definition of an integrated semantics for UML was summarized in [39]. In the following, we summarize the semantics in a less formal way.

4.2.1 Data Semantics

Our data semantics handles instances as tuples whose members are referred to by the corresponding attribute. The data semantics (i) defines the concept of *data state* which encapsulates the set of instances and the actual valuation of attributes and (ii) defines functions to obtain the value of an attribute in the context of an instance and for changing the valuation of an attribute in an instance. See the paper in appendix and MOGENTES Deliverable D3.2b for more details.

4.2.2 State Machine Semantics

Based on previous experience with modelling [38], model checking [30], code [35] and test generation [31] and runtime verification [36, 37] based on state machine models, we propose the extension of a previously introduced semantics with the notion of time events as outlined above.

The formal semantics introduced in [38] [36] divides UML state chart elements into two disjoint sets: *core concepts* are the fundamental modelling features of finite state-transition systems (e.g., states, transitions, guard predicates, activities, state refinement, etc.) while *advanced concepts* are the ones that do not conceptually extend the language just serve as convenience facilities (e.g., history vertices). The semantics focuses on the core concepts and provides transformation rules for substituting advanced concepts with structures built up

of core elements. In contrast to the UML standard the semantics unambiguously defines concepts related to state refinement, state containment and rigorously defines well-formedness requirements.

With respect to *activities* of state charts (associated to leaving and entering states and effects of transitions) the formalism uses *directed acyclic graphs* to unambiguously express subsequence relations amongst activities and indicating possibilities for parallel execution. Since the standard also lacks the definition of *compound transition structures* (i.e., ones built up of multiple transitions connected by various pseudo-state vertices) the concept of *transition conglomerates* representing transition structures of any complexity is introduced.

Formally speaking, the semantics of SMTE models is defined by a *Kripke transition system* (KTS). A KTS is a K=(S, L, T) triple above the AS set of status labels and AT set of transition labels, where S is the set of KTS states, $L : S \to A_S$ is the state labelling function and $T \subseteq (S \times AT \times S)$ is the labelled transition relation. The $s \in S$ states of the KTS represent statuses of the state chart. Here *status* is a composite concept involving (i) the actual configuration of the state machine, (ii) the phase of operation (idle, in an RTC step, terminated, etc.), (iii) the actual evaluation of variables and (iv) the actual value of clocks (A_S labels are tuples of this information assigned to states by L). The T labelled transition relation indicates the steps between statuses of the state chart where A_T transition labels indicate (i) the event whose reception made taking this edge (may be empty), (ii) the evaluation of guards that is necessary for taking this edge and (iii) the activity structure to be performed when taking this edge. See the paper in appendix and MOGENTES Deliverable D3.2b for more details.

4.2.3 Specifying Activiites

Another deficiency of UML is the lack of a programming language-like notation for the unambiguous specification of activities' bodies. In order to overcome this issue we proposed the introduction of the Activity and Guard Specification Language (AGSL), a high abstraction level programming language targeted for being embedded in UML models. AGSL enables the straightforward definition of guards, activity and method bodies. AGSL features a clean syntax and a formally defined semantics which is integrated to the data semantics (access to instance attributes) and the state machine semantics (communication by signals). See the paper in appendix and MOGENTES Deliverable D3.2b for more details.

4.3 Conversion of UML SMTE to Timed Automata

Timed Automata (TA) [14] are the underlying formalism of the model checker tool UPPAAL [13, 12]. UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types (bounded integers, arrays, etc.) thus a viable candidate for our goals in MOGENTES (see also Deliverable D 1.2). The query language of UPPAAL, used to specify properties to be checked, is a subset of *computation tree logic* (CTL).

Transforming UML SMTE models to Timed Automata promises several benefits, e.g., visual simulation, evaluation of various dependability related criteria (expressed in CTL) and test case generation [23]. As the semantics of SMTE is defined by a Kripke transition system, for their transformation to Timed Automata will involve the following straightforward steps:

- The UML model is automatically transformed to an intermediate formalism where the static structure is transformed to an easy to process form, AGSL code fragments are parsed and state machines are represented by Kripke transition systems (KTS). A KTS is a finite state-transition system whose states and transitions are labeled by labeling functions. In our case states of the KTS represent statuses of the UML state machine. The status concept was introduced by us encompassing (i) the actual configuration of the state machine (active states) and (ii) the phase of the run-to-completion behavior. Transitions of the KTS represent steps between statuses described by (i) the event that triggered the step (if any), (ii) the set of composite transitions that are fired and (iii) the composite activity structure that was performed (specified by a directed acyclic graph of activities for maximal flexibility). The intermediate formalism also holds the relevant parts of the static structure (packages, classes, etc.) thus the intermediate model can be seen as a self-contained pre-processed image of the input UML model.
- The intermediate model may also contain information about the environment of the system under modeling which may be specified by the modeler. We can indicate here which events may hit which instances

from the external environment and define the maximal allowed length of the event queue. This information can be automatically derived from the model if using the corresponding UML profile or can be provided manually in the intermediate model's editor.

- Finally the intermediate model is automatically transformed to an UPPAAL model. The model can be simply loaded into UPPAAL without any further human intervention.

The resulting model is ready for simulation and (after defining the corresponding requirements) for model checking or test case generation. The most important concepts of the UPPAAL model are the timed automata. For each class with an associated state machine the code generator constructs an automaton and there are three extra timed automata automatically added to the model:

- The Event Queue Template automaton is responsible for the management of the event queue and enforcing a run-to-completion behavior by synchronization with automata corresponding to user classes (mentioned above). The event queue template implements a FIFO event queue policy.
- The Deadline Observer Template automaton is responsible for the management of clocks and inserting time events into the queue at appropriate instances in time. There is one clock variable defined for each "deadline situation" instance i.e., those situations where a transition is triggered by a time event in the context of an object.
- The Environment Template automaton is responsible for inserting events coming from the environment into the event queue.

4.4 Test Case Generation Based on UML SMTE

A functional test case should demonstrate the fitness of the system for a particular purpose, e.g., a test case may aim at demonstrating that the system can be started, can achieve its normal working state and is able to deliver service upon users' requests. Additionally we may require that a given test campaign meets various coverage criteria (e.g., all states of the system traversed, all transitions performed etc.).

Automatic test generation for UML SMTE may be based on the idea discussed in [22, 15, 21]: (i) the SMTE model of the system should be *transformed* to a network of Timed Automata, (ii) *negations of test goals* should be specified as CTL expressions (e.g., the system is unable to answer a given user request) and (iii) UPPAAL should be instructed to *show a counterexample* for the intentionally faulty "requirement". It is easy to see that the environment's behaviour in the counterexample trace will be the *specification of the test case* (i.e., the test driver should behave as the environment behaved in the example trace). When having to meet various *coverage criteria* (e.g., state coverage) additional Boolean variables and corresponding actions can be introduced into the model (e.g., introducing a variable indicating that the given state has been entered previously and adding actions that set this variable to true on all edges entering that state). UPPAAL can be instructed to search for the shortest counterexample (thus the test case with the least number of steps) or the fastest counterexample (thus the test case that needs the less time to be processed).

If more sophisticated coverage criteria or testing support is need, there are various COTS tools available built on UPPAAL, e.g., UPPAAL-TRON (a testing tool suited for black-box conformance testing of timed systems, mainly targeted for embedded software) or CoVer (a tool for creating test suites from UPPAAL models with coverage specified by observer automata).

As discussed in MOGENTES Deliverable 1.2, the black-box conformance testing implemented with UPPAAL is consistent with Tretmans' untimed input-output conformance relation "ioco" [44].

4.5 Summary

The discussion above has presented our proposal for the semantic foundations of behavior modeling in MOGENTES, introduced an embeddable programming language for the definition of guards and activities and outlined the process of automatic test case synthesis on the basis of UML models. Our approach involves (i) defining a *formal semantics* for the SMTE models, (ii) a straightforward method for *translating* SMTE models to a network of Timed Automata, the input language of the UPPAAL model checker, finally we outlined the way of

	Survived Faulty Impl.	Detection Rate	Block Coverage	No. TCs
A2	2 / 38	95%	99%	128
A3	0 / 38	100%	99%	287
A4	0 / 38	100%	99%	129
A5	1 / 38	97%	99%	63
A6	0 / 38	100%	99%	11
A7	1 / 38	97%	99%	3
A8	13 / 38	66%	88%	9

Table 4: Overview of how many faulty SUTs could not be killed by the different test suites.

model checking and (iii) automatic *test case generation*. Our approach is built on well-established formalisms (UML, Timed Automata) and tools (UML modelling environments, the UPPAAL model checker); some extra information necessary for test case synthesis (i.e., specification of the environment) which not directly available in UML models was implemented by an UML profile.

5 Coverage and Mutations

The content of this section is covered in the technical annex [4], [17], and [40].

The quality of test suites is commonly measured in terms of coverage criteria (cf. [32]). One code coverage criterion, for example, might be to demand that each condition within if-statements of the program evaluates once to true and false. This kind of coverage is also called decision coverage and one of the many possible code coverage criteria. A few other examples are *function coverage* (each function has to be called), *statement coverage* (each line of the source code has to be executed), or *condition coverage* (also called predicate coverage; every Boolean sub-expression needs to be true and false).

Usually, increasing coverage indicates an increasing test suite quality: A test suite that is able to achieve full decision coverage is obviously better than one where just a few of the many decisions of the program are covered. Within MOGENTES, model-based mutation testing is employed for test case generation. Faults are injected into our system models and our test case generation tools create test cases that are able to reveal the injected faults. Hence, the question about the relation between code coverage criteria and the mutation techniques arises.

In [32], the relation between mutations and code coverage is explored. The notion of weak mutations is used to show that common condition coverage techniques are subsumed by mutation testing. Our small empirical study on the relation between code coverage and mutation coverage supports this result [4]. We have implemented Ford's CAS and by mutating the source code, we derived 38 faulty implementations, which were used to assess the quality of the different test suites generated by applying the approaches A2 to A8 already described in Section 3.5. Table 4 summarizes the results. For each test case generation approach, it gives an overview of the number of survived faulty implementations, the fault detection rate, the basic block coverage[2] on the original implementation, as well as the number of executed test cases.

In summing up, the results show that our approaches are powerful in detecting faults and show a high coverage on the implementation. However, the depth of our conformance analysis is critical as too less depth results in missing test cases and, in this example, in undetected faults. The results also show that the 3500 test cases of the first approach were by far too many: for the given model mutations, one path per mutation is sufficient to detect all faults, provided this path is long enough. Finally, the combined approach (A6) proved to be a nice trade-off between generation time and effectiveness.

Regarding our qualitative approach, we defined coverage criteria for qualitative reasoning (QR) models in [17]. Besides (re)definitions for *state-* and *transition coverage* in the QR domain, the notion of *domain coverage* (test cases that explore the complete qualitative domain, i.e., all qualitative values), *delta coverage* (test cases that explore the complete domain of QR value changes, i.e., +,0,-), and *complete delta coverage* (combination of domain and delta coverage) are introduced. Although not presented in the context of action systems, these coverage criteria are applicable to qualitative action systems as well.

[2] Basic block coverage is defined as the coverage of a sequence of bytecode instructions without any jumps or jump targets.

5.1 Mutation Selection

An important issue in mutation-based test case generation is the selection of mutations (see also Deliverable 4.1). The problem of choosing an appropriate combination of faults or a stronger fault can be addressed by defining a partial order over combinations of faults that orders them with respect to their strength (as suggested in [27]). Let μ and ν be mutations or faults. According to Definition 1 in [27], μ is *at least as aggressive* as ν if for every model M and every specification $\mathcal{S}$, we have that if M_μ adheres to $\mathcal{S}$, then so does M_ν. For instance, let μ be a fault that produces the effect that a certain binary signal changes non-deterministically. This fault is stronger than a fault ν that results in the same signal being permanently 1, since the possible behaviours induced by μ also contain the behaviour resulting from ν.

Note furthermore that a combination of two arbitrary faults μ and ν is *not* necessarily stronger than the single fault μ, since the fault effects might cancel each other out. The aggressiveness ordering introduced over results is a lattice. Its bottom element is the empty set of mutations and its top element represents the most aggressive mutation, which is possibly a combination of several mutations. The intention is to find a combined mutation in the lattice that is strong enough to yield a different output. Since the size of the lattice grows exponentially with the number of potential mutations and faults, a good heuristic must be chosen. Searching the lattice by beginning from the extremes (top or bottom) is certainly not a good strategy for two reasons: (1) The closer a combined mutation is to the top element of the lattice, the less likely it is to occur in reality since this would mean that *all* the mutations occur *simultaneously*. (2) The closer a combined mutation is to the bottom element of the lattice, the less likely it will have an impact on the output since the (combined) mutation might be too weak and might not propagate to the output.

This suggests a binary search on the lattice as proposed by Purandare et al. [40]. Here, the search starts from the middle of the lattice. If a combined mutation is found not to propagate to the output, all weaker mutations can be deleted form the lattice, since a weaker mutation will not affect the output either. In such a case, the search tries to find a stronger mutation. Otherwise, the algorithm outputs the current combined mutation or tries to find a weaker one. Purandare et al. [40] apply a similar technique for checking whether specifications vacuously hold. As observed by Kupferman et al. [27], this work is closely related to our problem. That is, the problem of finding mutations that violate a specification is dual to the problem of vacuity checking.

Another question related to mutation techniques is the issue whether one is able to reliably asses the quality of a test suite by simulating faulty programs with mutation techniques. For example, the work in [8] investigates this relation and concludes that *generated mutants can be used to predict the detection effectiveness of real faults*.

6 Conclusion

We have given an overview of the theoretical foundations we have worked on in the MOGENTES project. This research resulted into thirteen papers that form the technical annex of this deliverable.

The main areas of research in this deliverable are semantics, conformance, test case generation techniques and coverage. We defined a precise UML semantics by mapping it to action systems and timed automata, extended action systems with object-oriented features and qualitative reasoning techniques. In order to link action systems to the established testing theories, we interpret action systems as labelled transition systems. With this semantics at hand, we chose Tretman's input-output conformance relation ioco as a basis for test case generation.

Model-based mutation testing is used to generate test cases from action systems. Counter examples of input-output conformance form the basis for test cases. ioco can deal with simple timeout behaviour. However, for more complex timed behaviour, we rely on the Uppaal test case generation tools.

Coverage in this project is mostly interpreted from a fault-covering point of view: mutation coverage measures how many mutants can be killed. We discussed several approaches to select test cases, presented results of their efficiency to detect implementation faults and compared mutation to simple forms of code coverage. In addition, the selection of mutation operators was discussed.

Most of this fundamental research was translated into tools and evaluated in case studies. Hence, it is difficult to draw a clear line between theoretical and applied research. Therefore, this deliverable is closely related to

Deliverable 4.3 that concentrates on mutation-based test case generation and to Deliverable 3.1a that deals with fault models.

We have discussed limitations of our test case generation techniques and to overcome them is an ongoing effort as is always in model analysis. The most promising approaches seem to cleverly combine the complete conformance checking techniques with random exploration. By doing so, we have achieved great timing improvements.

As future work, we are going to explore symbolic execution techniques and the application of semantic mutations. If a lot of data is involved, symbolic techniques usually outperform explicit data enumeration. Therefore, future work will include the substitution of explicit data enumeration by symbolic techniques similar to our work in [25]. Semantic mutation analysis generates mutants that represent semantic variations of a modeling language. For example, there are several possible interpretations of UML 2 state machines. A translator would generate mutants of this semantic choices and generate test cases that would detect the behavioural consequences in an implementation, if any. This form of mutation testing would help the software engineers to fix the semantics of an ambiguous modeling framework as UML.

7 Technical Annex

The following publications are part of the technical annex and are shortly described in this section.

A Report on QR-Based Testing [18]. This paper talks about the basic concepts needed for QR-based test case generation. In particular *test purposes*, and an input-output conformance relation based on *ioco* are presented. Using these results, the authors then present an example of how to apply the theory in practise.
A test purpose, after Jard and Jeron, describes some aspect of the specification that is of interest for testing. It is defined as a regular expression over symbols that represent properties of model quantities. During test case generation, the original specification is replaced by the result of the synchronous product of test purpose and original specification. In other words, test purposes are used to cut out an interesting sub-part of the specification and, hence, counter state-space explosion issues. For further details, please see the attached paper.

Coverage-Based Testing Using Qualitative Reasoning Model [17]. In this paper the authors define coverage criteria for QR models that can be used for test case generation. Besides (re)definitions for *state-* and *transition coverage* in the QR-domain, the authors introduce the notion of *domain coverage* (test cases that explore the complete qualitative domain, i.e., all qualitative values), *delta coverage* (test cases that explore the complete domain of qr-value changes, i.e., +,0,-), and *complete delta coverage* (combination of domain and delta coverage). All these coverage criteria are used together with *test purposes* to generate test cases. The authors then compare the different coverage criteria in terms of transitions covered.

Qualitative Action Systems [6]. This work sets the foundations for the theory of qualitative action systems. Based on the established theories of hybrid action systems and qualitative reasoning, the authors first formalise the abstraction of continuous functions to qualitative functions. Based on this formalisation the authors then define data refinement between qualitative and hybrid action systems before presenting an example system.

Qualitative action systems extend ordinary action systems by so called *qualitative actions* of the form $e_q :\rightharpoonup d_q$ where e_q is a so called evolution guard and d_q is a set of qualitative differential equations, describing the continuous evolution of the system when the evolution guard is satisfied.

Unifying Input Output Conformance [3]. The conformance relation ioco is currently our choice of conformance relation in the MOGENTES project. In this paper we are reexamining ioco and reformulate it in a denotational, predicative semantics. The benefits of this new theory can be summarised as follows: (1) Instead of describing the assumptions of ioco informally, the new formalisation presents the underlying assumptions as unambiguous healthiness conditions and by adopted choice operators over reactive processes; (2) Our formalisation naturally relates ioco and refinement in one theory; (3) The denotational version of ioco enables formal,

machine-checkable, proofs. (4) Due to the predicative semantics, test case generation based on the presented theory can be seen as a satisfiability problem. This facilitates the use of modern sat modulo theory techniques (SMT) for test case generation. (5) Finally, this version of ioco broadens the scope of ioco to specification languages with similar semantics, e.g., to generate test cases from Action System specifications.

Strengthening Properties Using Abstraction Refinement [40]. Based on the notion of vacuity, which essentially means that a formula holds for unintended reasons, e.g., left hand side of an implication $\rightarrow$ is never true, and within the context of model checking LTL properties, the authors present an algorithm that succeeds in exposing *vacuity to the user in the form of a strengthened and shortened formula* that preserves satisfaction. This is done by a search in a lattice of candidate formulae in combination with an instance of the counterexample-guided abstraction refinement technique. To put it differently, counterexamples provided by the model checker help in narrowing down the search space in the lattice of candidate solutions.

Formal Model-Based Mutation Testing with UML: Techniques and Results [4]. This paper presents the techniques and results of a novel model-based test case generation approach that automatically derives test cases from UML state machines. Mutation testing is applied on the modeling level to generate test cases. This work presents the test case generation approach, discusses the tool chain and presents the properties of the generated test cases. The main contribution of this paper is an empirical study of a car alarm system where eight different approaches are compared. Detailed figures on the effectiveness of the test case generation strategies are included. Although, UML serves as an input language, all techniques are grounded on solid foundations.

Model-Based Mutation Testing of Hybrid Systems [5]. This paper presents a novel model-based testing approach to test embedded systems controlling a continuous environment, i.e., hybrid systems. Two key abstractions, against which conformance is systematically checked, are employed: (1) Classical action systems are used to model the discrete controller behavior. (2) Qualitative differential equations are used to model the evolutions of the environment. The latter is based on a technique from the domain of Artificial Intelligence called qualitative reasoning. Mutation testing on these models is used to generate effective test cases. A test case generator has been developed that searches for all test cases that would kill a mutant. The mutant models represent the used fault models. The generated test cases are then executed on the implementation in order to systematically exclude the possibility that a mutant has been implemented.

Mapping UML to Labeled Transition Systems for Test-Case Generation - A Translation via Object-Oriented Action Systems [26]. This paper extends on the formalism of object-oriented action systems (OOAS) and describes a mapping of a selected subset of UML to OOAS by choosing one of the several possible semantics of UML. This mapping, together with the introduction of a trace semantics for OOAS, paves the way for applying tools for and theory of labeled transition systems to UML models. As a running example, a car alarm system in the context of model-based test case generation is used to illustrate how the UML mapping is done.

Automated Conformance Verification of Hybrid Systems [19]. This work presents a new approach for verifying the input-output conformance of two hybrid systems. This approach can be used to generate mutation-based test cases. In this paper, hybrid systems are specified within the framework of Qualitative Action Systems. Here, besides conventional discrete actions, the continuous dynamics of hybrid systems is described with so-called qualitative actions. This paper then shows how labeled transition systems can be used to describe the trace semantics of Qualitative Action Systems. The labeled transition systems are used to verify the conformance between two Qualitative Action Systems. Finally, first experimental results on a water tank system are presented.

Conformance Testing of Hybrid Systems with Qualitative Reasoning Models [1]. This paper proposes the use of qualitative models, which are an abstraction of quantitative physical models, for test case generation and test execution. In particular, it shows how Simulink models from which control programs are automatically

extracted can be tested with respect to qualitative models. Since Simulink models are heavily used in industry, the approach is of practical interest.

Fault-Based Conformance Testing in Practice [45] This work reports on experiences and findings when applying fault-based conformance testing to an industrial application. Besides a discussion on modeling and simplifications, it presents a technique that prevents an application from implementing particular faults. Faults are modeled at the level of the specification. We show how such a technique can be adapted to specifications with large state spaces and present results obtained when applying our technique to the Session Initiation Protocol and to the Conference Protocol. Finally, we compare our results to random and scenario-based testing.

When BDDs Fail: Conformance Testing with Symbolic Execution and SMT Solving [25]. This work presents a new symbolic test case generation technique. Symbolic methods usually outperform explicit data enumeration if a lot of data is involved. The new approach is based on symbolic execution and on satisfiability (modulo theory; SMT) solving. This work was motivated by the complete failure of a well-known existing symbolic test case generator to produce any test cases for an industrial Session Initiation Protocol (SIP) implementation. Hence, the BDD-based analysis of the existing tool has been replaced with a combination of symbolic execution and SMT solving. The new tool generates the test cases for SIP in seconds. However, further experiments showed that this approach is not a substitutive but a complementary approach: the technique and the results obtained for two protocol specifications are presented, the first supporting the new technique, the second being witness for the classic BDD technique.

Test Case Generation Based on an Integrated UML Data and Behavior Semantics [39]. This paper presents an *integrated semantics* for UML and an *executable language* for specifying activity bodies. Based on the solid foundations of the composite semantic, it presents a method for automatically generating test cases for a real-time safety critical system.

References

[1] Bernhard Aichernig and Harald Brandl. Conformance testing of hybrid systems with qualitative reasoning models. In *MBT 2009, the 5th Workshop on Model Based Testing*, 2009.

[2] Bernhard Aichernig, Willibald Krenn, Henrik Eriksson, and Jonny Vinter. MOGENTES deliverable D1.2 State of the art survey - part a: Model-based test case generation, 2008.

[3] Bernhard Aichernig and Martin Weiglhofer. Unifying input output conformance. In *UTP 2008, the 2nd International Conference on Unifying Theories of Programming*, 2008. informal proceedings.

[4] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Formal model-based mutation testing with UML: Techniques and results. Technical report, Institute for Software Technology, Graz University of Technology, 2010. Submitted to ICFEM 2010.

[5] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Model-based mutation testing of hybrid systems. In *Proceedings of Formal Methods for Components and Objects (FMCO) 2009*, 2010. in print.

[6] Bernhard K. Aichernig, Harald Brandl, and Willibald Krenn. Qualitative action systems. In *11th International Conference on Formal Engineering Methods, ICFEM*, pages 206–225, 2009.

[7] Bernhard K. Aichernig, Harald Brandl, and Franz Wotawa. Conformance testing of hybrid systems with qualitative reasoning models. *Electron. Notes Theor. Comput. Sci.*, 253(2):53–69, 2009.

[8] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[9] Ralph-Johan Back, Luigia Petre, and Ivan Porres. Continuous action systems as a model for hybrid systems. *Nordic Journal of Computing*, 8(1):2–21, 2001.

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

[11] Ralph-Johan Back and Kaisa Sere. Superposition refinement of parallel algorithms. In *FORTE '91: Proceedings of the IFIP TC6/WG6.1 Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 475–493, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[12] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[13] Gerd Behrmann, Alexandre David, Kim G. Larsen, and Wang Yi. A tool architecture for the next generation of UPPAAL. In *UNU/IIST 10th Anniversary Colloquium. Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 352–366, 2003.

[14] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, pages 87–124. Springer, 2004. LNCS 3098.

[15] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing*, pages 125–139, 2005. LNCS 3395.

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

[17] Harald Brandl, Gordon Fraser, and Franz Wotawa. Coverage-based testing using qualitative reasoning models. In *Proc. of the 20th International Conference on Software Engineering and Knowlegde Engineering (SEKE)*, pages 393–398, 2008.

[18] Harald Brandl, Gordon Fraser, and Franz Wotawa. A report on QR-based testing. In *22nd International Workshop on Qualitative Reasoning*, pages 1–9, 2008.

[19] Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. Automated conformance verification of hybrid systems. In *QSIC*, 2010. in press.

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

[21] Anders Hessel, Kim Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, pages 77–117. Springer, 2008. LNCS 4949.

[22] Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using UPPAAL. In *Proc. of the 3rd International Workshop on Formal Approaches to Testing of Software*, pages 1100–1100, 2004.

[23] Anders Hessel and Paul Pettersson. A global algorithm for coverage-based test case generation. In *Proc. of the 3rd Workshop on Model-Based Testing*, 2007.

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

[25] Elisabeth Jöbstl, Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. When BDDs fail: Conformance testing with symbolic execution and SMT solving. *Software Testing, Verification, and Validation, 2008 International Conference on*, pages 479–488, 2010.

[26] 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 *Proceedings of Formal Methods for Components and Objects (FMCO) 2009*, 2010. in print.

[27] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–9. IEEE, 2008.

[28] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[29] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, 1997.

[30] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, December 1999.

[31] I. Majzik, G. Pintér, and Z. Micskei. Development of model based tools to support the design of railway control applications. In F. Saglietti and N. Oster, editors, *Proc. 26th Int. Conf. on Computer Safety, Reliability and Security, SAFECOMP 2007, Nuremberg, Germany*, volume LNCS 4680, pages 430–435. Springer Verlag Berlin, 2007.

[32] A. Jefferson Offutt and Jeffrey M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical Report ISSE-TR-96-01, George Mason University, 1996.

[33] OMG. A UML profile for MARTE: Modelling and analysis of real-time embedded systems, 2008. Beta 2, OMG Adopted Specification.

[34] OMG. OMG Unified Modelling Language (OMG UML), 2009. Version 2.2.

[35] G. Pintér and I. Majzik. Automatic code generation based on formally analyzed UML statecharts. In G. Tarnai and E. Schnieder, editors, *Formal Methods for Railway Operation and Control Systems (Proceedings of Symposium FORMS-2003, Budapest, Hungary, May 15-16)*, pages 45–52. L' Harmattan, Budapest, 2003.

[36] G. Pintér and I. Majzik. Run-time Verification of Statechart Implementations. In Cristina Gacek, Alexander Romanovsky, and Rogerio de Lemos, editors, *Architecting Dependable Systems*, pages 148–172. Springer-Verlag, 2005.

[37] G. Pintér and I. Majzik. Error detection in control flow of event-driven state based applications. In P. Pelliccione, H. Muccini, N. Guelfi, and A. Romanovsky, editors, *Software Engineering of Fault Tolerant Systems*, volume Ser. Software Engineering and Knowledge Engineering, Vol. 19. World Scientific Publishing, 2007.

[38] Gergely Pinter. *Model Based Program Synthesis and Runtime Error Detection for Dependable Embedded Systems*. PhD thesis, Budapest University of Technology and Economics, 2007.

[39] Gergely Pintér and István Majzik. Test case generation based on an integrated uml data and behavior semantics. Technical report, Budapest University of Technology and Economics, 2010. Submitted to FORMS/FORMAT-2010.

[40] Mitra Purandare, Thomas Wahl, and Daniel Kroening. Strengthening properties using abstraction refinement. In *Proceedings of DATE 2009*, Nice, France, 20/04/2009 2009.

[41] Mauno Rönkkö, Anders P. Ravn, and Kaisa Sere. Hybrid action systems. *Theoretical Computer Science*, 290:937–973, 2003.

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

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

[44] Jan Tretmans. Model based testing with labelled transition systems. In R.M. Hierons, J.P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of Lecture Notes in Computer Science, pages 1–38. Springer, 2008.

[45] Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. Fault-based conformance testing in practice. *Int. J. Software and Informatics*, 3(2-3):375–411, 2009.

[46] Tomi Westerlund and Tiberiu Seceleanu. An UML profile for action systems. Technical report, Turku Centre for Computer Science, 2003.