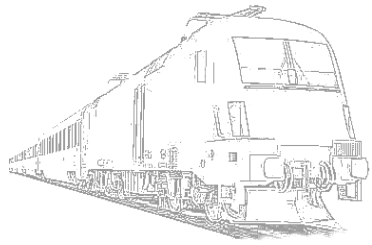
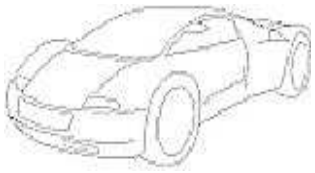
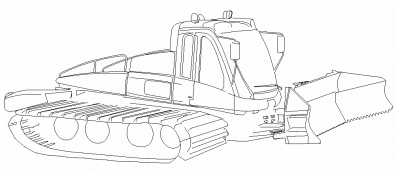
MOGENTES

Model-Based Generation of Test-Cases for Embedded Systems

State of the Art Survey

Part a: Model-based Test Case Generation Techniques

Project	MOGENTES		**Contract Number**		216679
Document Id	1-19a_1.1r	**Date**	2008-06-30	**Deliverable**	D 1.2
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 team members				

Change History

Version	Date	Reason for Change	PagesAffected
0.1w	2008-05-26	first version (internal)	all
0.2w (rev. 106)	2008-05-30	added lots of material (internal)	all
0.3d (rev. 132)	2008-06-10	word style, first draft	all
0.4d (rev. 138)	2008-06-13	updated word style, fixes	all
1.0r (rev. 152)	2008-06-30	added Matelo, Dotgear, fixes, British-English	all
1.1r (rev. 162)	2008-10-22	added list of abbreviations	all

Executive Summary

Model-based test generation techniques are on the rise. This is not surprising because they promise cheap, flexible and correct test cases in a world of ever increasingly complex systems. Because model-based testing draws from formal methods and the wide body of known testing techniques, a lot of scientific work exists. This survey gives an overview of recent advances in the field. The large number of references (> 200) to related work demonstrates how active this research area is. Hence, the survey does not claim completeness, but shows the main trends and advances with respect to test case generation from models.

The survey starts by giving an overview of the possible modelling techniques relevant for automated test case generation. The reason is that a model-based testing technique mainly depends on the style of its models: (1) The models determine the level of abstraction and by this, the granularity of the testing strategy. (2) Different technical challenges arise for different modelling styles. For example, the proper sequencing of interactions with the system under test is no problem in Finite State Machine models, because these sequences are explicitly represented in the transition relation. However, for contract-like specifications proper sequences must be calculated by checking the compatibility of a method's postcondition with the precondition of its successor. As a consequence, the model-based testing techniques cannot be separated from the modelling techniques.

First, we discuss contract-like specifications that use relations on the state variables for specifying what a method is supposed to do. These contracts can be used as test oracles, analysed with respect to test equivalence classes, and explored for proper test sequences. For executable versions of such models (model programs), traditional white-box techniques can be applied.

Next, we review abstract data types and their related testing techniques. These models are in general more abstract than contracts, but are harder to produce. The reason is that a sound and complete set of axioms needs to be found stating the behaviour of a class (module) by relating the effects of its methods. However, once these axioms are produced, it is relatively easy to derive test cases from them. For details we refer to a recent survey dedicated to this testing technique.

Labelled transition systems (LTS) are the next kind of models for model-based testing to be presented. An LTS model is automaton-like with the testing stimuli represented on the transitions (as labels). They may be infinite and non-deterministic. Several conformance relations (e.g. allowing incomplete models) have been proposed, we give a brief overview and discuss tools.

Another class of recent work relies on model checkers. These tools check temporal properties of models and produce counterexamples for refutation. The counterexamples are traces of computations through the system. These traces have been used as test cases for deterministic systems, including real-time systems.

Next, general work on test case generation from finite state machines is discussed. This has a longer tradition than model checking and deserves a separate review, although we focus on recent tools.

Data-flow models and hybrid models conclude our discussion on models and test case generation. Data-flow models focus on the flow of data between software components, like in electronic circuits. Relations can be used to specify how these streams of data are processed. This is highly related to hybrid systems, where discrete and continous state models are intermixed. Test case generation from hybrid systems is still in its infancy.

After the model-oriented discussion, we switch to fault injection, an orthogonal technique with respect to test case generation. Its main purpose is to evaluate and debug error handling mechanisms.

Finally, to complete the picture, some recent industrial case-studies applying model-based testing are reviewed. The relatively low number of prominent industrial case studies indicates the urgent need to make the presented techniques more amenable to industry. To be precise, automated test case execution with models is widely applied, but test case generation from models still lacks industrial acceptance.

Contents

1 Introduction

In spite of several decades of research, guaranteeing the quality of a software product still represents a major practical duty for the industry from both points of view of results and costs [201]. Solutions from research in software engineering have been provided that, from most of the industry's perspective, are either too complex to apply, or too obtrusive in their development process, or both. Between the two extremes – zero quality assessment and rigorous inspection with formal methods – the maximum part of the industry has shown to settle for testing so far.

With that in mind, MOGENTES aims at significant improvement of testing in industrial context by developing methods and tools for automatic generation of efficient test cases. In particular, such test cases shall be derived from appropriate models of the respective systems to be tested.

1.1 Scope and Purpose

In order to achieve the ambitious MOGENTES goals, it was recognised that a sound knowledge of the recent developments and state of art for model-based test case generation (MBTCG) is crucial. Therefore, one of the first activities within MOGENTES was a survey about this topic; its results are presented in this document. As guidance for structuring both the survey and the document, following considerations have been applied.

With testing, a system or some part of it is executed with a set of selected stimuli, and observed to determine whether its behaviour conforms to a specification or an expectation. Testing is expected to both reveal defects, and to increase the confidence in their absence when no defects manifest anymore.

As a quality assessment activity, testing is sound, as observed misbehaviours are necessarily defects, but it is not complete unless all the possible stimuli are explored. Due to combinatorial explosion this is not a real option and the effectiveness of testing is bound to the capability of selecting *significant* stimuli. In practice this activity is still performed manually: Stimuli are commonly decided directly from the code, chosen randomly, or generated from heuristics that were built relying on human expertise.

Testing can be performed with a *white-box* or *black-box* strategy, depending on whether the actual implementation of the software is used to select the stimuli or whether it is intentionally ignored to focus on its external behaviour. Both strategies have strengths and weaknesses, and they usually coexist and are employed differently at different levels of abstraction. A mixture of white and black-box testing is called *grey-box* testing.

Model-based testing mostly is a black-box variant of testing in which the selection of the stimuli is performed at a high level, exploiting a specification of the expected behaviour of the system from which test cases can be automatically generated. The possibility of deriving test cases from a formal specification has the additional benefit of providing a more structured, repeatable process for selecting test cases and without relying on subjective experience. The availability of a formal specification also eases the process of building an oracle.

Model-based testing requires the definition of a suitable model, the generation of the actual test suite, and the actual execution of the generated cases.

The remainder of the document is organised as follows. The fundamental terminology about testing is presented in the next Paragraph 1.2. Section 2 presents the modelling paradigms and languages available. Section 3 surveys techniques for generating test cases. Before presenting case studies in Section 5, we give an overview of fault injection techniques in Section 4.

1.2 Terminology in Testing

A software system can be evaluated for adequacy with respect to a specification or with respect to the expectations of its stake-holders. The first activity is referred to as *verification* while the second is *validation*. In this context, testing is concerned with the former case.

The actual software whose quality is under assessment is referred to as *System Under Test* or *Implementation Under Test*, often shortened *SUT* and *IUT* . The SUT is run on a processor and may be surrounded by further software such as an operating system; these represent the *SUT environment*. During a *test* the system is executed

and observed; its behaviour varies depending on its stimuli, for example input data and events from the environment. The part of these stimuli that is relevant for the test is called *test input*. A determined behaviour is expected from the system during a test with respect to the test input, which is referred to as *test output*. A *test case* is the combination of the input and output part of a test. A *test suite* is a collection of test cases. If test cases are not directly executable, they are called *abstract* and so is called their test suite. Abstract test cases stem, for example, from generation from models that abstract the implementation under test. Executable test cases may be specified as *concrete*.

In general, the set of test inputs in a test suite must be sorted out from an intractably large set of all the possible inputs for a software. This selection is driven by a *test selection criterion*, such as the coverage of transitions or functionalities from the requirements document. The formal counterpart of a selection criterion is called *test case specification* or *test purpose*, and provides an operative definition for how to extract cases.

The selection of a small set of test cases with respect to all the possible inputs can be justified through the introduction of *test selection hypotheses*. Assuming that "all the inputs in a determined range cause equivalent behaviour with respect to correctness" is one such example. Gaudel [98] frames these hypotheses as the point of contact between testing and proof, as they enable to reduce the number of tests to perform, but require to be proven to guarantee formal correctness [46].

The test case specification defines a test suite. When the test suite is abstract, test cases must be made concrete before executing the tests; this may involve any of the input and output parts of the test case. The function that maps abstract test cases to concrete ones is referred to as *adapter*.

When a test is executed, the actual output of the SUT must be compared with the expected output indicated in the test case; the adapter is also in charge of comparing the concrete and abstract values at this stage. The comparison produces a *test verdict* indicating the result of the test, ranging over *pass*, when the concrete output matched the expected abstract one, *fail* when it did not match, or *inconclusive* when the adapter cannot compare them. The general problem of determining whether an output is correct with respect to a specification is known as *the oracle problem*.

1.3 Partners Contributions

This document has been produced by TUG, SP, and ETH, with contributions from ARC and other partners.

2 System Models for Testing

This section gives a brief overview of different techniques to model the system under test. It also presents important conformance relations that are used later on by the test case generation tools in order to decide whether a given implementation conforms to the specification.

We start with *contract-like specifications* for modelling the effects of individual system operations of an API . Then, we discuss *abstract data-type* models, a more abstract way of defining an API via the algebraic properties of its operations. We then start with behavioural specifications, such as *process algebras*, *labelled transition systems*, *temporal logic*, *EFSMs and State Charts* , and models for *hybrid systems*.

Within all mentioned categories, different methods for modelling systems exist. Some of them will be more abstract than others, so the level of abstraction of the resulting tests and also the systems that can be modelled may differ within modelling approaches of one category.

2.1 Contract-like Specifications

Contract-like specifications became prominent with the advent of *design-by-contract*, but the concept is known since the early 70-ies. Their common feature is that an explicit model of the state is used to express what an operation expects from its environment and what it has committed to compute. Therefore, this languages are also known as state-based specification languages. Formally, the contracts of an API are a set of predicates over a set of state variables as partial relations between system states. In this context, 'state' denotes the valuation of a set of variables according to an equivalence class. The equivalence classes are computed from the preconditions and postconditions of operations by using constraint solving techniques. The states are abstract and can represent an infinite number of concrete states. In contrast to FSMs or transition systems where states are represented explicitly contract-like specifications use implicit states that are computed as required.

State variables can be sets, sequences, relations, and functions. Operations introduce changes to the system state variables. If the precondition of an operation holds prior to its execution then it is guaranteed that the operation's postcondition holds after execution. Pre/post conditions are first order predicates over state variables and according operations define a partial relation between system states.

There are a lot of examples for contract-like specification languages:

- VDM [146], Z [140], B [2]
- CIRCUS [176]
- ALLOY [142]
- Eiffel (Design-By-Contract) [167], SPARK [19], Spec# [20], Java[1]-Modelling Language (JML) [48], ...
- OCL [215]
- PiSPEC

In the following we provide a short example of a system, modelled in Z. See section 5 for a sketch of a B model.

Z The language Z uses schemas to structure a specification. The following example from [127] shows the Z schema of a *stack* and the according *pop* operation:

$$
\begin{array}{|l}
\hline
Stack \\
items : seq\ Object \\
\hline
\#items \leq maxSize \\
\hline
\end{array}
$$

[1] In [94] the authors give an overview of formal specification languages for Java.

The items on the stack are a *sequence* of type *Object* and the size is bounded to *maxSize*. The *pop* operation changes the state of the stack denoted by $\Delta Stack$:

$$
\begin{array}{|l}
\hline
Pop \\
\Delta Stack \\
x! : Object \\
\hline
items \neq \langle\rangle \\
items' = tail\ items \\
x! = head\ items \\
\hline
\end{array}
$$

If the precondition, i.e. the stack is not empty, holds then one item is removed from the stack and is returned in the output variable *x*. In Z names ending in ! denote output and names ending in ? denote input. For the *pop* operation *x* is an output of type *Object*. The operation's precondition requires the stack to be not empty. The postcondition states that after the operation the stack contains all elements but the head. The head is assigned to the output parameter *x*. Contract-like specifications offer a means to describe systems in a declarative manner, similar to functional or logic programming languages.

Spec# Microsoft's SpecExplorer [211] builds on Spec# and distinguishes between input (controllable actions) and output (observable actions) for testing. Hence, in order to detect conformance of a program, it needs a conformance relation that knows about input and output: "Alternating simulation" provides this distinction and was chosen as conformance relation for SpecExplorer. We discuss the tool in Section 3 in detail.

2.1.1 Alternating Simulation

Alternating simulation was introduced in [10] as a refinement relation between alternating transition systems that can be used to model composite systems. Each transition in an alternating transition system corresponds to a possible move in a game between components.

An informal definition of alternating simulation can be found in [72]: Consider two systems P and Q. *Alternating simulation is a relation between the states of P and the states of Q such that, at related states, all the outputs that can be generated by P can also be generated by Q, and all the inputs that can be accepted by Q can be accepted by P; moreover, corresponding inputs and outputs lead to states of P and Q that are again related.*

Based on alternating simulation, it is possible to define a notion of refinement: Informally, the system P refines Q, if there exists an alternating simulation between the initial states. Having this refinement relation that is based on compatibility of input assumptions and output guarantees, [72] builds the theory for a modelling framework for component-based design and verification. It is outside the scope of this paper to go into the very details.

2.2 Abstract Datatypes

Another model for test case generation are abstract datatypes (ADTs) . In this algebraic specification technique an object class or type is specified in terms of the relationships between the operations defined on that type. Unlike in model-based specifications, nothing is said about the internal representation of the defined objects. Hence, in general, algebraic specifications are more abstract than model-based specifications. Formally, an ADT consist of a signature Σ and positive conditional equations as axioms, where the signature declares sorts and sorted function symbols.

For example, a container of natural numbers can be specified as follows:

$$
\begin{aligned}
\textbf{Sorts} &: \mathit{Container}, \mathit{Nat}, \mathit{Bool} \\
\textbf{Functions} &: \\
\mathit{empty} &: \mathit{Container} \\
\mathit{add} &: \mathit{Nat} \times \mathit{Container} \to \mathit{Container} \\
\mathit{isin} &: \mathit{Nat} \times \mathit{Container} \to \mathit{Bool} \\
\mathit{remove} &: \mathit{Nat} \times \mathit{Container} \to \mathit{Container} \\
\textbf{Axioms} &: \\
\forall\, x, y : \mathit{Nat}, c : \mathit{Container} &: \\
& \mathit{isin}(x, \mathit{empty}) = \mathit{false} \\
\mathit{eq}(x, y) = \mathit{true} &\Rightarrow \mathit{isin}(x, \mathit{add}(y, c)) = \mathit{true} \\
\mathit{eq}(x, y) = \mathit{false} &\Rightarrow \mathit{isin}(x, \mathit{add}(y, c)) = \mathit{isin}(x, c) \\
& \mathit{remove}(x, \mathit{empty}) = \mathit{empty} \\
\mathit{eq}(x, y) = \mathit{true} &\Rightarrow \mathit{remove}(x, \mathit{add}(y, c)) = c \\
\mathit{eq}(x, y) = \mathit{false} &\Rightarrow \mathit{remove}(x, \mathit{add}(y, c)) = \mathit{remove}(x, c)
\end{aligned}
$$

Here, the ADT specification of the container uses the ADTs Bool and Nat, the latter providing the equality operation eq.

Algebraic specification was brought to prominence by Guttag [111] in the specification of abstract data types. Various notations for algebraic specification have been developed, including OBJ [92] , Larch [113, 112] and CASL [171]. In RAISE [107, 108, 70], algebraic specification and the model-based techniques from VDM have been combined. A notation independent presentation of the subject can be found in [134]. In the process algebra Extended Lotos (ELOTOS) [138] the data part is specified in terms of ADTs.

The main idea of test case generation from ADTs is to unfold the axioms and test if the operations representing the function symbols satisfy the axioms.

2.3 Process Algebras

Process algebras, which are so-called behavioural specifications, are used to describe the observational behaviour of systems. They are well suited to model distributed systems because parallel processes may communicate via synchronisation of selected events. Languages for process algebras are, e.g. LOTOS (Language of Temporal Ordering Specifications) [138], CCS (Calculus of Communicating Systems) [168], CSP (Communicating Sequential Processes)[130]. An application area for process algebras is the specification of protocols, e.g., in the telecommunication sector. The authors in [6] describe the protocol conformance testing of a SIP registrar specified with LOTOS.

2.4 Labelled Transition Systems

Labelled Transition Systems (LTS) are related to process algebras because often LTS are used to describe the semantics of a process algebra.

A Labelled Transition System (LTS) is commonly defined by a tuple $\mathit{LTS} := \{S, s_0, \Sigma, \delta\}$ where S denotes a set of states, s_0 is the initial state, Σ is the alphabet of labels, and $\delta : S \times \Sigma \to S$ defines a state transition relation. Examples of modelling languages with LTS semantics are π-Calculus [169], LOTOS [138], SDL [141] , and Promela [133]. The difference to finite state machines is that an LTS may have an infinite number of states and transitions.

Inspired from I/O automata and CSP, the authors of [195] define an input-output-symbolic transition system (IOSTS) to enable symbolic test generation. The theory gives rise to the STG [61] tool. Informally, an IOSTS is a transition system comprising a nonempty finite set of locations and transitions between these locations. A state is a

tuple consisting of a location and a valuation for the variables and parameters. A transition models either an input, output, or internal action. Each transition consists of one location describing origin and one location describing the destination of the transition, a boolean expression (guard), a set of expressions – called the assignment of the transition –, an action, and a tuple of messages (the messages sent/received in the action). A transition can be fired if its guard is true and its action synchronises. It then updates the state by performing the specified assignments.

An IOSTS can be instantiated by providing values for parameters. An instantiated IOSTS can be initialised by providing an initial condition that assigns a value to each variable.

Another extension of labelled transition systems are *action machines* [105], which combine concepts of process algebra, abstract state machines, and finite automata: They extend LTS with *the necessary notions to deal with sequential composition in the presence of rich data state: namely accepting states, which determine when to transition from one to the next machine; and initialisation transitions, which allow propagating the environment (data part) to the next machine*. The authors of [105] say that action machines are more general than IOLTS because they allow composition of arbitrary models for the purpose of slicing.

There exist many conformance relations for labelled transition systems [205]. In the following we present some of the more relevant ones for testing.

2.4.1 CONF

In contrast to classical conformance relations like *testing preorder* or *trace preorder* [204] the *conf* [204] relation only deals with traces that are part of the specification. This makes it applicable for testing, since the large complement of traces that are not in the specification do not need to be considered.

The conf relation states that for all traces in the specification the observational behaviour of a trace in the implementation including deadlock behaviour must be a subset of the same trace in the specification. This means that beside trace inclusion the same deadlocks can be observed in both the implementation and the specification.

The input-output variant of the conf relation is called *ioconf*. In the following subsection we discuss *ioco* , a more fine-grained conformance relation with respect to *ioconf*.

2.4.2 IOCO

The input-output conformance relation (ioco), as presented in [206, 205], relates input-output labelled transition systems (IOLTS) and can be used to test the conformance of an implementation and a specification. More precisely, an implementation under test (IUT) and a specification (S) are ioco conform, iff the outputs of the IUT are outputs of S after an arbitrary suspension trace[2] of S.

In difference to ioconf, the ioco relation is able to distinguish behaviour traces that differentiate in time-outs (quiescence) only.

A couple of variations of ioco have been defined [206]. These variations include definitions of symbolic ioco [87], hybrid ioco, real-timed ioco, and multi ioco. Note that in difference to [205] in recent work [206] test cases are redefined to be input enabled.

Similar to [7] a formal definition of ioco can be given as follows.

Definition 1 *An input output labelled transition system (IOLTS) is a labelled transition system* $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ *with* Q^M *a finite set of states,* A^M *a finite alphabet (the labels) partitioned into three disjoint sets* $A^M = A_I^M \cup A_O^M \cup \{\tau\}$ *where* A_I^M *and* A_O^M *are input and output alphabets and* $\tau \notin A_I^M \cup A_O^M$ *is an unobservable action,* $\rightarrow_M \subseteq Q^M \times A^M \times Q^M$ *is the transition relation and* $q_0^M \in Q^M$ *is the initial state.*

We use the following classical notations of labelled transition systems for IOLTSs. Let $q, q', q_i \in Q^M$, $Q \subseteq Q^M$, $a_{(i)} \in A_I^M \cup A_O^M$ and $\sigma \in (A_I^M \cup A_O^M)^*$: If the tuple (q, a, q') is element of the transition relation $\rightarrow_M$, we write $q \xrightarrow{a}_M q'$. Similar, if there exists a q' so that the tuple (q, a, q') is in $\rightarrow_M$, we write $q \xrightarrow{a}_M$.

[2] trace with quiescence

The shorthand for saying that $q = q'$ or that there is a trace of unobservable τ-actions leading from q to q' is $q \stackrel{\epsilon}{\Rightarrow} q'$. Writing $q \stackrel{a}{\Rightarrow} q'$ means that there is a $q1$ and $q2$, so that $q \stackrel{\epsilon}{\Rightarrow}_M q_1 \stackrel{a}{\rightarrow}_M q_2 \stackrel{\epsilon}{\Rightarrow}_M q'$. This can be generalised to arbitrary sequences of actions σ, in which case we write $q \stackrel{\sigma}{\Rightarrow} q' = q \stackrel{a_1 \ldots a_n}{\Rightarrow} q'$ and that is defined as as $\exists\, q_0, \ldots, q_n : q = q_0 \stackrel{a_1}{\Rightarrow}_M q_1 \ldots q_{n-1} \stackrel{a_n}{\Rightarrow}_M q_n = q'$.

We denote q **after**$_M$ $\sigma =_{df} \{q' \mid q \stackrel{\sigma}{\Rightarrow}_M q'\}$ and Q **after**$_M$ $\sigma =_{df} \bigcup_{q \in Q}(q$ **after**$_M$ $\sigma)$. We define $Out_M(q) =_{df} \{a \in A_O^M \mid q \stackrel{a}{\rightarrow}_M\}$ and $Out_M(Q) =_{df} \bigcup_{q \in Q}(Out_M(q))$. $Traces(M)$ denotes all possible sequences of actions $\sigma \in (A_I^M \cup A_O^M)^*$. We will omit the subscript $_M$ (and superscript ^{M}) when it is clear from the context.

An IOLTS M is *weakly input enabled* if it accepts all inputs in all states, possibly after internal τ actions: $\forall\, a \in A_I^M, \forall\, q \in Q^M : q \stackrel{a}{\Longrightarrow}$. An IOLTS is *deterministic* if for any trace there is at most one successor state, i.e., $\forall \sigma \in \left(A_I^M \cup A_O^M\right)^* :\mid q_0^M$ **after**$_M \sigma \mid \leq 1$, where $\mid X \mid$ denotes the cardinality of the set X. An IOLTS is *complete* if it allows all actions in each state, i.e. $\forall\, q \in Q^M, \forall\, a \in A^M : q \stackrel{a}{\longrightarrow}_M$.

Commonly the symbol δ is used to represent quiescence. A quiescent state is a state, that has no edge labelled with an output or an unobservable action. Thus, $q \stackrel{\delta}{\rightarrow} q$ means, that q is a quiescent state. To define the ioco relation we need the suspension automaton, which makes quiescence observable by considering δ as an output.

Definition 2 *The suspension automaton of an IOLTS $S = \left(Q^S, A^S, \rightarrow_S, q_0^S\right)$ is an IOLTS $\Delta(S) = (Q^S, A^{\Delta(S)}, \rightarrow_{\Delta(S)}, q_0^S)$ where $A^{\Delta(S)} = A^S \cup \{\delta\}$ with $\delta \in A_O^{\Delta(S)}$. The transition relation $\rightarrow_{\Delta(S)}$ is obtained from $\rightarrow_S$ by adding loops $q \stackrel{\delta}{\rightarrow} q$ for each quiescent state. The traces of $\Delta(S)$ are called the suspension traces of S and are denoted by $STraces(S)$.*

For the ioco relation, we assume that the behaviour of an implementation can be expressed by an IOLTS. The following definition of the ioco relation says, that an implementation I conforms to a specification S, iff the outputs of I are outputs of S after an arbitrary suspension trace of S.

Definition 3 *Let S be an IOLTS and I be an weakly input enabled IOLTS, where the alphabets of I and S are compatible, i.e., $A_I^S \subseteq A_I^I$, and $A_O^S \subseteq A_O^I$, then*

$$I \textbf{ ioco } S \quad =_{df} \quad \forall\, \sigma \quad \in \quad Straces(S) \quad : \quad Out_I(\Delta(I) \textbf{ after } \sigma) \quad \subseteq \quad Out_S(\Delta(S) \textbf{ after } \sigma).$$

2.5 Kripke Structures and Temporal Logic

Gordon Fraser et al. present in [180] a survey about testing with model checkers. A model checker is a tool used for formal verification. It takes as input an automaton describing a system and a temporal logic property. Then it searches the model state space in order to determine if there is a violation of the stated property. If so the model checker returns a counter example, i.e., a path from an initial state to the state where the property violation occurs. Model checkers can proof that certain properties of a model hold and return a counter example otherwise. The idea of testing with model checkers is to use counter examples as test cases. Therefore the test objective is formulated as negated temporal property in order to get counter examples that satisfy the property.

The formalism commonly used to describe model checking and to define the semantics of temporal logics is the Kripke structure.

Definition 4 (Kripke Structure) *A Kripke structure K is a tuple $K = (S, S_0, T, L)$:*

- *S is a finite set of states.*
- *$S_0 \subseteq S$ is an initial state set.*
- *$T \subseteq S \times S$ is a total transition relation, that is, for every $s \in S$ there is a $s' \in S$ such that $(s, s') \in T$.*
- *$L : S \rightarrow 2^{AP}$ is a labelling function that maps each state to a set of atomic propositions that hold in this state.*

AP is a countable set of atomic propositions.

In the following we define traces and paths over Kripke structures. An infinite execution sequence of this model is a *path*. A Kripke structure defines all possible paths of a system.

Definition 5 (Path) *A path $p := \langle s_0, s_1, ... \rangle$ of Kripke structure K is an infinite sequence such that $\forall\ i \geq 0 : (s_i, s_{i+1}) \in T$ for K.*

Let $Paths(K, s)$ denote the set of paths of Kripke structure K that start in state s. We use $Paths(K)$ as an abbreviation to denote $\{Paths(K, s) \mid s \in S_0\}$.

As infinite paths are not usable in practice, model checking uses finite sequences, commonly referred to as *traces*. If necessary, we can interpret a finite sequence as an infinite sequence where the final state is repeated infinitely.

Definition 6 (Trace) *A trace $t := \langle s_0, ...s_n \rangle$ of Kripke structure K is a finite sequence such that $\forall\ 0 \leq i < n : (s_i, s_{i+1}) \in T$ for K. There can be a dedicated state s_i such that $s_i = s_n$ and $i \neq n$, which is a loopback state, and $\langle s_0, \ldots s_{i-1}, (s_i \ldots s_n)^\omega \rangle$ is a path of K.*

A trace t is either a finite prefix of an infinite path or a path that contains a loop, if a loopback state is given. The latter is called a lasso-shaped sequence, and has the form $t := t_1(t_2)^\omega$, where t_1 and t_2 are finite sequences. The sequence t_2 is repeated infinitely often, denoted with ω, the infinite version of the Kleene star operator used for ω-languages. Lasso shaped sequences are used in practice to show violation of liveness properties, which requires infinite sequences. For example, the model checker NuSMV [59] interprets all identical states in a trace as possible points of loopback. The number of transitions a trace consists of is referred to as its *length*. For example, trace $t := \langle s_0, s_1, ..., s_n \rangle$ has a length of $length(t) = n$.

In model checking properties over Kripke structures are formulated in temporal logic which is most commonly Linear Time Logic [184] (LTL) or Computation Tree Logic [62] (CTL) . Temporal logics are model logics with special operators for time. In comparison to propositional logic which can only reason about states temporal logics can reason about paths and trees. CTL*, introduced by Emerson and Halpern [81], is the superset of these logics. Most current model checkers support either LTL or CTL, or sometimes both. Other temporal logics that are used in model checking are Hennessy-Milner Logic [122] (HML) , Modal μ-calculus [156], and different flavours of CTL such as timed [8], fair [82], or action [173] CTL.

Time operator in LTL formulas are:

- "$\bigcirc$" referring to the *next* state. E.g. "$\bigcirc\, a$" denotes that a has to be true in the next state.
- The *until* operator "U" means that for a formula "$a\ \mathsf{U}\ b$" a has to hold from the current state up to a certain state where b holds.
- The *always* operator "$\Box$" states that in "$\Box\, a$" a is true for all states of a path starting from the current state.
- "$\Diamond$" means that a property *eventually* holds in the future.

The LTL syntax and semantics is defined as follows:

Definition 7 (LTL Syntax) *The BNF definition of LTL formulas is given below:*

$$\begin{aligned} \phi \ ::= \ & true \mid false \mid a \in AP \mid \neg\, \phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \\ & \phi_1 \rightarrow \phi_2 \mid \phi_1 \equiv \phi_2 \mid \phi_1\ \mathsf{U}\ \phi_2 \mid \bigcirc \phi \mid \Box \phi \mid \Diamond \phi \end{aligned}$$

Definition 8 (LTL Semantics) *Satisfaction of LTL formulas by a path $\pi \in Paths(K)$ of a Kripke Structure*

$K = (S, S_0, T, L)$ is inductively defined as follows, where $a \in AP$:

$$K, \pi \models true \quad \text{for all } \pi \tag{1}$$
$$K, \pi \not\models false \quad \text{for all } \pi \tag{2}$$
$$K, \pi \models a \quad \text{iff} \quad a \in L(\pi_0) \tag{3}$$
$$K, \pi \models \neg\phi \quad \text{iff} \quad K, \pi \not\models \phi \tag{4}$$
$$K, \pi \models \phi_1 \wedge \phi_2 \quad \text{iff} \quad K, \pi \models \phi_1 \wedge K, \pi \models \phi_2 \tag{5}$$
$$K, \pi \models \phi_1 \vee \phi_2 \quad \text{iff} \quad K, \pi \models \phi_1 \vee K, \pi \models \phi_2 \tag{6}$$
$$K, \pi \models \phi_1 \rightarrow \phi_2 \quad \text{iff} \quad K, \pi \not\models \phi_1 \vee K, \pi \models \phi_2 \tag{7}$$
$$K, \pi \models \phi_1 \equiv \phi_2 \quad \text{iff} \quad K, \pi \models \phi_1 \text{ iff } K, \pi \models \phi_2 \tag{8}$$
$$K, \pi \models \phi_1 \ \mathsf{U} \ \phi_2 \quad \text{iff} \quad \exists\, i \in \mathbb{N}_0 : K, \pi^i \models \phi_2 \wedge \forall\ 0 \leq j < i : K, \pi^j \models \phi_1 \tag{9}$$
$$K, \pi \models \bigcirc \phi \quad \text{iff} \quad K, \pi^1 \models \phi \tag{10}$$
$$K, \pi \models \Box \phi \quad \text{iff} \quad \forall\, j \in \mathbb{N}_0 : K, \pi^j \models \phi \tag{11}$$
$$K, \pi \models \Diamond \phi \quad \text{iff} \quad \exists\, j \in \mathbb{N}_0 : K, \pi^j \models \phi \tag{12}$$

Clarke and Emerson [62] introduced CTL which is a subset of CTL*. CTL* formulas, introduces by Emerson and Halpern [81], extend propositional formulas with the temporal operators (**F** , **G** , **U** , **R** , **X**) and additional path quantifiers (**A**, **E**).

Model checking is divided into three approaches. First model checkers used *explicit model checking*. Here the state space is represented explicitly and searched by forward exploration until a property violation was discovered. *Symbolic model checking* is based on ordered binary decision diagrams (BDDs) [47] . BDDs serve as an efficient representation of states and relations between them. The performance of this approach depends on the number of BDD variables and their order. *Bounded model checking* [165] expresses the underlying problem as a constraint satisfaction problem (CSP) . This enables the use of SAT solvers to calculate counter examples up to a certain bound.

The most commonly used model checkers in the context of testing are the explicit state model checker SPIN [132] (Simple Promela Interpreter), the Symbolic Analysis Laboratory SAL [73] , which supports both symbolic and bounded model checking, the symbolic model checker SMV [153] as well as its derivative NuSMV [59] , which supports symbolic and bounded model checking.

Testing with model checkers exploit counter examples as test cases. This approach can be classified as testing with test purposes because the formulation of a temporal property refers to a test purpose. A test purpose selects some behaviour in the specification which someone wants to test and can be formulated as never-claims [83] or as partition of the execution tree [52].

2.6 (E)FSM, State Charts

Modelling languages based on the finite states paradigm represent *reactive systems* through states, that capture the internal status of the system, and transitions, that let the system switch from one state to another in reaction to a determined stimuli like input data or events. Transitions can also produce actions, such as output or messages.

Finite state models have been frequently investigated for model-based test case generation in the past, especially for checking properties of protocols and standards. The GSM mobile communication system is a popular case study of protocol checking [23], while [104] shows the use of test-generation tools for checking a standard.

Finite state based notations are of particular interest for theory, because their limited expressiveness makes them especially easy to process automatically, and at the same time they are relevant in practice, because their simplicity makes them easy to learn and to adopt in practice. Petrenko et al. discuss conformance relations for FSMs in [183].

Several actual languages exploit this paradigm. The well known Finite State Machines, State Charts [117], UML State Machines [193] , and Extended Finite State Machines [55] are some examples. Of these, the fundamental language of Finite State Machines is introduced as a reference, and UML State Machines are presented more extensively as an example of practical use.

A Finite State Machines (FSM) is formally represented by a tuple

$$(S, i, \Sigma_i, \Sigma_o, R, b)$$

where S is the set of states, $i \in S$ is the initial state of the system (without loss of generality it can be unique), Σ_i and Σ_o are the input and output alphabet, respectively, R is the transition relation: $R \subseteq S \times \Sigma_i \times S$, and b is the behaviour function $b : S \times \Sigma_i \to \Sigma_o$.

FSMs are often represented graphically through a set of nodes and arrows connecting them. Figure 1 shows the FSM specifying a simplified withdrawal use case of an ATM . States are labelled for further understandability. The system starts from state "ready", and correct executions of the system must terminate such that the machine is ready again for another operation.

This graphical form is convenient for readability and intuitiveness for humans, but it has the severe disadvantage of becoming intractable when the size of the specification grows beyond few states. Referring to the previous example, introducing the ability to withdraw any amount of money would result in an explosion of the size of the diagram. Most of the effort of the extensions to this language concentrate in mitigating this problem.

UML State Machines [193] is a popular notation based on finite states notation that has become part of the Unified Modelling Language family; it is based on State Charts from which it inherits most of its features.

Figure 2 shows an example for the same ATM withdrawal operation expressed as a State Machine. The State Machine diagram is significantly more compact and readable than the FSM version. In UML State Machines, the primary facility that allows to reduce the size of diagrams is the hierarchy of machines.

Through hierarchy it is possible to collapse several system states into one that represents the "invariant" of the former process. In the example, the ATM withdrawal behaviour is sliced in three tiers: levels of depth the process `insertCard` and `checkPin*` can be synthesised in one single, more abstract state `Authentication`. The language allows to specify actions that can be performed when a state is entered and exited, and possibly actions that last as long as the system is in the state.

The notation also supports variables, so that it is possible to simplify states and transitions replicated for different values of a parameter. In the ATM example, this can be applied for flattening the `want*` states, with their respective incoming and outgoing transitions, into one `want[amount]` state with the corresponding `press(amount)` and `provide(amount)` transitions.

Transitions can be annotated with guards and actions, with the general form *stimulus[guard]/action*, where stimulus is an event like a function call, guard is a predicate that must hold for running the transition, and action is one or more operations caused by running the transition. These further data can be expressed in several notations, including the Object Constraint Language [190]. This adds a significant expressiveness to the language, although increasing its complexity.

The possibility of reducing the state space allows to simplify systems to any extent. The main goal of modelling with notations based on the finite states paradigm is defining a set of states that at the same keeps a uniform level of abstraction, and makes the specification neither too detailed nor compact to the point of being difficult to get.

Extended Finite State Machines is a further language based on finite states, which is a simpler extension to FSMs with respect to UML State Machines. In EFSM [55], transitions can be bound to a set of predicates. As soon as all the predicates are true, the transition can fire. Predicates are defined with respect to a set of machine variables, and transitions are allowed to change these values.

2.7 Hybrid Systems

Hybrid systems comprise discrete and continuous behaviour and can be expressed, e.g., with hybrid automata. They can also be modelled with languages like χ [162] and CHARON [9] which provide compositional modelling. The work in [192] presents an extension of *action systems* [14] to *hybrid action systems*. They describe the parallel composition of two hybrid action systems where global variables are merged and local variables are hided. The composition is restricted to systems of linear differential equations.

Another formalism for hybrid systems provides the language *Modelica* [91]. It is an object oriented language for modelling hybrid systems with combined continuous and discrete time. It follows the data flow principle and

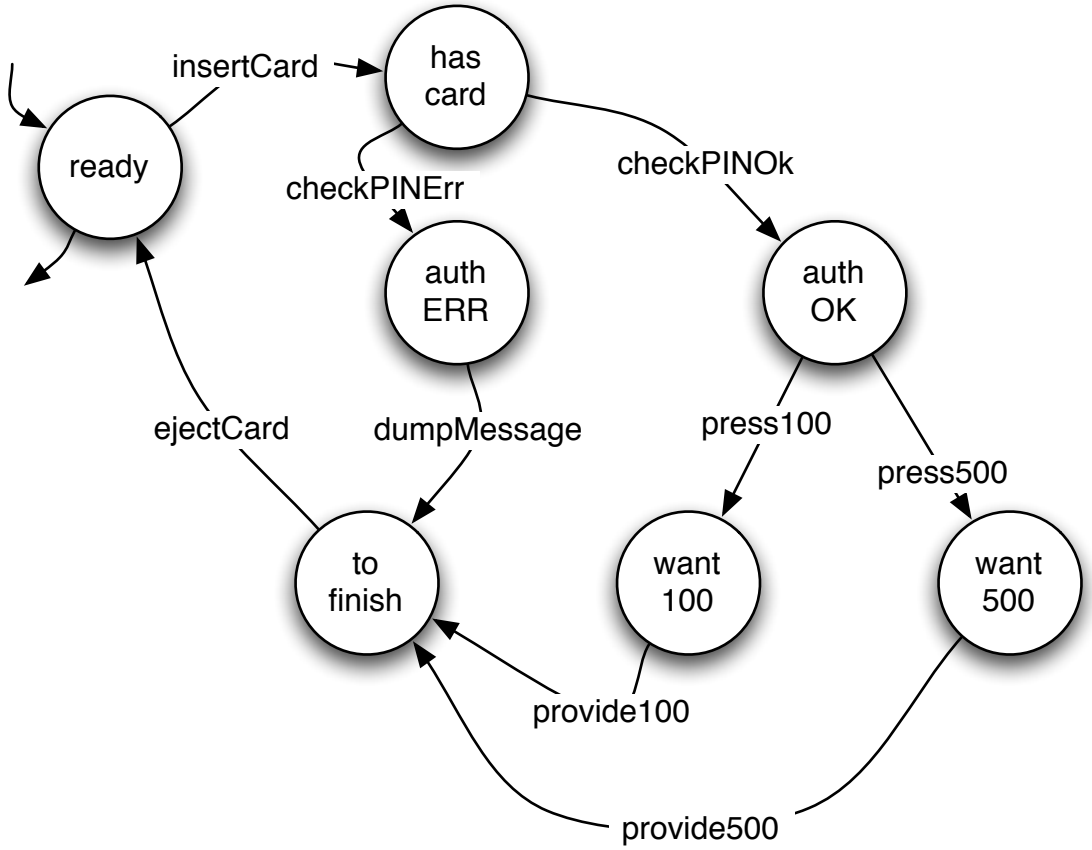


Figure 1: Finite State Model describing the behaviour of an ATM

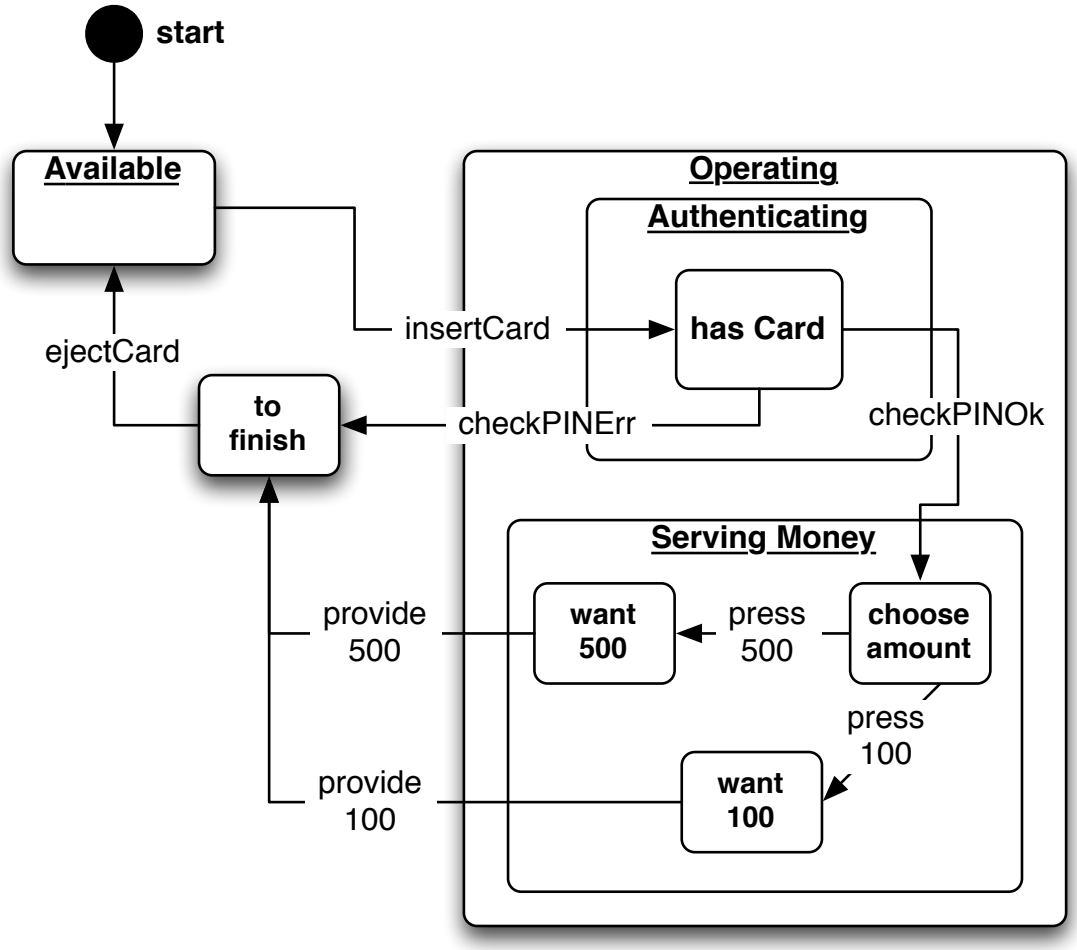


Figure 2: UML State Machine diagram for the ATM withdrawal

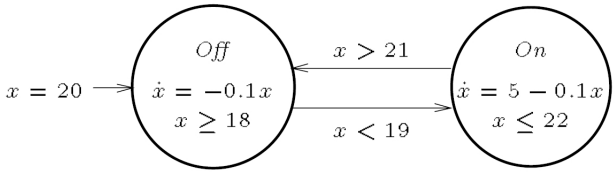


Figure 3: Hybrid Automaton describing a Temperature Controller. (Taken from [123])

synchronises continuous flows with discrete time events. In Modelica system components are specified via a set of variables and algebraic equations.

Hybrid automata [123] are finite state machines where the symbolic states, called *control modes*, can represent an infinite set of concrete states. States comprise assignments to variables, definition of state invariants, and *flow conditions*. Flow conditions are differential equations that describe the continuous behaviour of variables between state transitions. Figure 3 depicts the well known example of a temperature controller [123]. The variable x represents the temperature with 20°Celsius as initial value. In control mode *Off* the differential equation $\dot{x} = -0.1 \cdot x$ denotes a flow condition, i.e., the temperature x follows the function $e^{-0.1 \cdot t}$. Further the state contains the invariant $x \geq 18$ declaring that the state is left when the invariant is violated at the latest. The state can be left before as soon as the temperature falls below 19°Celsius. In the *On* control mode a heater causes the temperature to increase. The invariant of control mode *On* together with the guarded transition to control mode *Off* causes the heater to be turned off somewhere between 21°and 22°.

Timed automata are a subset of hybrid automata as they comprise one flow condition per clock variable. The flow condition describes the progress of time, i.e., $\dot{x} = 1$. Timed automata can be analysed with the model checking and testing tool UPPAAL, described in section 3.5.1.

2.7.1 HIOCO

Similar to the *ioco* testing theory the work in [209] introduces the conformance relation *hioco* for hybrid systems. It states that in every system state the discrete and continuous output behaviour of the implementation must conform to the hybrid specification. Transitions are labelled with actions that can be discrete or continuous. Continuous actions are called trajectories $\sigma \in \Sigma$ where $\sigma =_{def} (0, t] \rightarrow val(V)$ valuates a set of variables V. Further the set of actions is partitioned into input and output actions. In addition to the traditional definition of *ioco* the set of trajectories in the implementation is filtered due to input trajectories in the specification. So only relevant traces for the current control mode remain. The authors propose a test case generation algorithm which needs adjustments in order to be implementable.

2.7.2 Qualitative Reasoning Models

Qualitative Reasoning (QR) is an AI technique for modelling and reasoning about physical systems. The underlying theory is based on Qualitative Differential Equations (QDE) that are an abstraction of Ordinary Differential Equations (ODE) defined over the sign algebra [154]. A common QR modelling and simulation tool is *Garp3* by Bredeweg et al. [44]. Garp3 takes a system description comprising a set of model fragments and an initial scenario as input and generates a transition system as output. The transition system represents the development of model variables (quantities) over time. Figure 4 shows a Garp3 model fragment describing the behaviour of a battery in a declarative manner. A description of the Garp3 syntax and a user guide can be found in [37, 43]. Figure 5 depicts the generated transition system and Figure 6 shows the value history of model variables over time leading to an empty battery.

2.7.3 Matlab Simulink

Matlab Simulink is an IDE commonly used in industry for designing and simulating control systems, e.g., in the automotive area. The models can be hybrid containing discrete and continuous parts. When the simulation of a model meets the expectations one can use the *Realtime Workshop* code generator to get an implementation

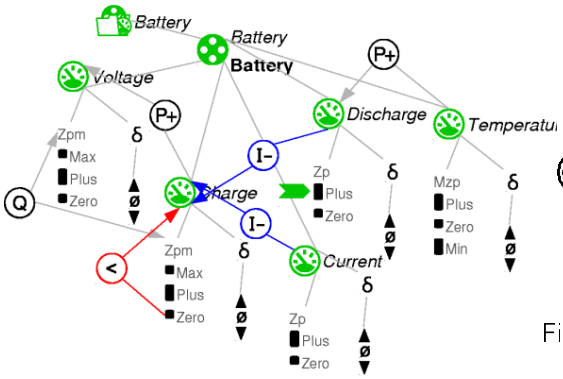


Figure 4: battery

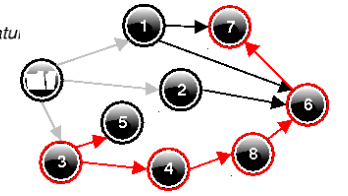


Figure 5: Battery State Chart

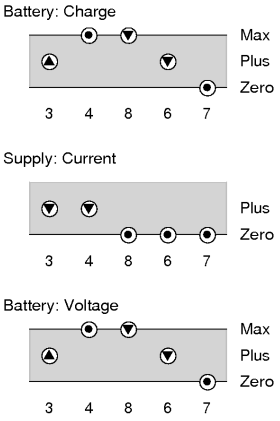


Figure 6: Battery Value History

with certifiable code[3]. Discrete components are modelled with *Simulink Stateflow*. It is also possible to discretise the continuous part by introducing sampling intervals. The authors in [53] present an approach to transform time-discrete Simulink models to the data flow language *Lustre* [115].

2.8 Dataflow Models

Data flow models describe how streams of data are processed by operators, e.g., two streams of integers are merged to a result stream $a_i + b_i = c_i$, $0 < i < n$. The operators are black boxes with inputs and outputs and process data synchronously. Dataflow architectures are well suited for distributed computing as data streams can flow between several processing units. Well known data flow languages are Lustre [115] and Esterel [25]. The tool *SCADE* by *Esterel Technologies* is based on Lustre and provides a code generator for certified *C* or *ADA* code. SCADE is used to specify safety critical systems and provides beside the code generator a graphical editor, a simulator, a design verifier, measurement for test coverage, and gateways to other tools like Simulink and UML/SysML [177].

[3] e.g., DO-178B Level A. No MathWorks products are qualified to DO-178B. For additional details see http://www.mathworks.com/support/solutions/data/1-10HP3P.html?solution=1-10HP3P

3 Test Case Generation Techniques

This section presents an overview of the state of the art in automated test case generation. Having introduced the possible ways of how to model a given system, we now turn our attention to the process of using the model together with some testing strategy and a conformance relation in order to extract test cases.

3.1 Testing Strategies

There are many testing strategies depending on the testing assumptions for an IUT. In the case of infinite state specifications testing assumptions enable the selection of a finite set of test cases in order to check if an implementation is correct regarding the specification. The stronger the assumption the fewer test cases are needed, ending in a formal proof. So having no testing assumptions requires complete testing to ensure correctness. The relationship between testing and proof is elaborated in [98].

Consequently, there are different strategies one can use to generate suitable test cases. In this part, we will introduce a classification of testing strategies that we follow within the next subsections.

Exhaustive Testing. The very first testing strategy we want to cover is *exhaustive testing*: According to the IEEE Std 2003–1997 , exhaustive testing can be defined as follows:

> Exhaustive testing seeks to verify the behaviour of every aspect of an element, including all permutations. For example, exhaustive testing of a given user command would require testing the command with no options, with each option, with each pair of options, and so on, up to every permutation of options. The various command options and permutations rapidly approach numbers too large to reach execution completion in realistic time frame. As an example, there are approximately 37 unique error conditions in POSIX.1 . The occurrence of one error can (and often does) affect the proper detection of another error. An exhaustive test of the 37 errors would require not just one test per error but one test per possible permutation of errors. Thus, instead of 37 tests, billions of tests would be needed (2 to the 37th power). Exhaustive testing is normally infeasible.[135]

Note that when dealing with finite, abstract models, exhaustive testing may be feasible: Starting with the finite model, abstraction will prune further states, removing details that contribute to combinatorial explosion.

Testing with Equivalence Classes Test input data form equivalence classes depending on the observations a tester can make. Consider a tester which observes a single output variable and only recognises if it is equal to zero or not. Then the input domain can be partitioned into two classes which behave equivalent in terms of observed outputs. Such abstract observations require strong testing assumptions that have to be proved. Without any testing assumptions testing with a completely enumerated input domain is required to ensure correctness. Testing assumptions are formulated as hypothesis, e.g., the uniformity hypothesis by Gaudel [98]. The uniformity hypothesis states that if the execution of a test case with input values according to an equivalence class leads to a pass verdict it will pass for all input values of this class.

Testing With Purposes. A second testing strategy, that tries to avoid the mentioned exponential growth, is to use *test purposes* that can be used to narrow down the specification to some smaller subset before generating test cases. A test purpose can be seen as a slicing criterion that cuts out the part of the specification which is of interest for testing. For example *TGV* computes the synchronous product of a test purpose with a specification resulting in a complete test graph (CTG) . The CTG contains all test cases satisfying the states test purpose. Other approaches use environmental models to restrict specifications. Such models are composed in parallel with the specification synchronising on common events. Non-shared events disappear as hidden leading to abstract states and possible nondeterminism.

Random Testing. As third testing strategy *random testing* also tries to avoid the exponential blow-up. This is done by exploring the specification randomly by selecting new events until a certain bound is reached or a test purpose is satisfied. Most approaches use random testing *on-the-fly (online)* during test case execution rather than precomputing test cases off-line. When dealing with nondeterministic specifications this has the benefit that test cases are adaptive, i.e., they evolve depending on received inputs from the environment. On the other hand online test cases are not reproducible.

Fault-Based Testing. Another testing strategy is *fault-based testing*: Most of the time it is possible to anticipate properties of a fault, for example that developers tend to use wrong variables in conditions. Fault-based testing takes advantage of this knowledge and tries to come up with test cases that can detect a certain fault when present. Fault-based testing often uses mutation of the correct program to generate the anticipated, faulty program version.

3.2 Testing from Contract-like Specifications

Assertions and Design-by-Contract are concepts well known by developers today. It can be attributed to this fact that there are a lot of tools and techniques available for automated test case generation that fit this particular modelling approach.

Equivalence Class based Testing Dick and Faivre [76] proposed a method of deriving test cases from VDM specifications by rewriting the pre/post conditions to a Disjunctive Normal Form (DNF) . Each disjoint case of an operation forms a sub-relation describing a single test domain. This activity is called partition analysis and the extracted sub-operations are used to compute a Finite State Machine (FSM). This FSM is required for test sequencing, i.e., find a sequence of operations that transfers the system into a state from which a desired operation can be tested. The aim for testing is to find sequences that cover all sub-operations of the FSM. Since the FSM is an abstraction of, in general, specifications with infinite data domains non-determinism can arise. When a new operation is added to a sequence the composition of resulting constraints are resolved. The new constraints can then be used to resolve, if possible, non-determinism. If it is not possible, the decision about the next operation in the sequence cannot be made until test execution. The authors describe a test case execution algorithm which consists of:

- finding appropriate test sequences;
- the selection of appropriate test data;
- the verification of the result of the operation;
- the comparison of the state of the physical system with abstract states in the FSM.

Random Testing *JET* [56, 93] performs random, dynamic testing of Java classes. It relies on a JML [48, 159] specification that is used as test oracle and for test case generation.

The tool automates all three major steps of program testing: Test data generation, test execution, and result determination. In all three steps, the method under test – including the JML annotations – plays an important role. In order to find a valid test case, JET repeatedly invokes the method until a set of arguments is found that satisfies the precondition. If the set of arguments satisfy the precondition, JET looks at the post condition. As the main idea of design-by-contract is that the postcondition holds when the precondition is satisfied, violation of the postcondition means finding an inconsistency between the implementation (method code) and the specification (JML annotation).

Test data generation as presented in [56] is a random process: For arguments of simple type (e.g. int), JET chooses a random value. For arguments of class type, JET creates an object of matching type and employs some mutation operations [4] according to some pre-determined, random sequence of method calls.

[4] A mutation operation is – overly simplified – a non-static method that returns void.

In [58] the authors improve the approach by building the sequence incrementally, which means to check whether the precondition is fulfilled before doing another random method call. It is reported that this improvement can find more meaningful test cases and produce longer call sequences.

In recent work [57] the authors extend random testing with constraint solving: By transforming the test data generation problem into a constraint satisfaction problem and adding randomness to the constraint solver's enumeration process, the authors achieve a vast improvement in time needed to create test data while not compromising data diversity.

Directed Testing (White-Box) *DART* [100] (Directed Automated Random Testing) uses a combination of concrete and symbolic execution to create test data that leads to the program pursuing different paths. During concrete execution, DART generates a conjunction of symbolic constraints along the path. In order to generate further test data that directs the execution flow along another path, these symbolic constraints are then modified and, if feasible, solved. If it is not feasible to solve the modified constraints, substitution by random values is used. DART does a depth first exploration of the paths by systematically negating conjuncts in the path constraint. DART is able to track linear integer arithmetic.

CUTE [198] builds upon the idea of DART and extends it with a method for solving approximate pointer constraints. While DART and CUTE work on C code, jCUTE [197] is an implementation for Java code.

Similarly, *EXE* [50] uses dynamic symbolic execution of C source code to find bugs. EXE supports bit-vector arithmetic, independent constraint optimisation, constraint caching, and tracking of indirect memory accesses symbolically. A recently proposed optimisation [33] targets the problem of the exponential number of paths in code.

Microsoft recently published *Pex* [203], another tool relying on concrete and symbolic execution for test data generation. Pex relies on the .NET framework and uses the Z3 [74] solver. Pex prioritises the search so a high coverage is achieved quickly. As is the case with CUTE, DART and EXE, Pex relies on instrumenting the program in order to do the symbolic execution in parallel. In particular Pex uses the .NET profiling API to inspect and rewrite the CIL [80] instructions of a method prior to just-in-time compilation. The instrumented code then drives a "shadow interpreter" that constructs and maintains symbolic representations and records conditions over which the program branches. The "shadow interpreter" thereby models the behaviour of all verifiable and most unverifiable .NET instructions. Recent work[68] builds on Pex to infer program invariants.

The idea of recording path conditions to create new test input has also been successfully used in white-box fuzz testing [101] of large applications (e.g. part of Microsoft Office 2007). It is also applicable in malware analysis, as shown by the authors of [170].

Although not primarily concerned with test generation, the recently published *DASH algorithm* [21] uses symbolic execution and test generation to proof programs in spirit of SLAM [17]. Thereby DASH uses test generation to guide where to perform the refinement of abstraction, and also to decide how the abstraction should be refined. The authors compare SLAM and DASH (both using Z3) on a set of 95 programs. While SLAM needs 20 minutes, DASH finishes all programs in 17 seconds. When the authors enabled test caching, it is reported that DASH outperforms SLAM with a factor of 300, taking only 4 seconds.

Multi Strategy (SpecExplorer) Relying on an extension of the alternative simulation refinement relation, *SpecExplorer* [211] from Microsoft takes a model program, which defines the state variables and update rules of an Abstract State Machine [109], explores the state space and extracts test cases from the model automaton (an extension of an interface automaton) built during exploration. SpecExplorer needs the Microsoft .NET Framework and uses Spec# [20], an extension of C#, as specification language.

SpecExplorer uses multiple strategies to control the test generation process:

1. For the exploration process, which builds the model automaton from the specification and is the first step in test-case generation, various restrictions can be given: The user can specify restrictions on the parameters used to call an action, actions can be disabled in states by setting the appropriate precondition, state filters can be used to prune the search space, bounds can be given for the exploration process, the search can also be directed, and the user can supply abstraction functions that group found states.

2. SpecExplorer knows two different modes for test-case generation: In the off-line case, SpecExplorer creates a test suite in form of a model automaton, while in the online case, SpecExplorer does not explicitly pre-compute a test suite but does a dynamic unfolding of the model program while running the test.

3. When generating test cases for a test suite, SpecExplorer knows different traversal algorithms, from Chinese postman tours, over shortest path algorithms, to random walks and algorithms based on Markov decision process theory.

4. During online testing, controllable actions are randomly chosen. The probability of choosing an action can be influenced with weights.

SpecExplorer relies on a model program as specification. Most important within a model program are methods marked with an *action* attribute: When reading the model, SpecExplorer will look for that kind of methods. Note that there are two different kinds of actions: *controllable* and *observable* ones. Controllable actions specify behaviour that can be invoked by a user, while observable actions are outputs from the system. In addition, each action has assigned a precondition that tells SpecExplorer in which state it is legal to use the action. Global variables store information about the current system state.

In order to derive valid action sequences from the model program, SpecExplorer uses an exploration algorithm to build a model automaton. Simply speaking, SpecExplorer in state s determines the set of enabled actions. Invoking an action from the set produces a set of updates that changes some state variables. Applying these updates to s produces the target state.

As indicated earlier, the model automaton is used to create test cases. To guarantee termination of test execution, the implementation under test is encapsulated in an observationally complete wrapper. During test execution, actual objects (created by the implementation) have to be bound to the model. Because these objects need to be considered in the conformance relation, alternating simulation is extended to take object bindings into account.

The authors provide a nice example of a chat system in [211]. For another discussion of SpecExplorer, see [208]. Finally, it is worth noting that SpecExplorer and Pex have been combined [152].

Multi Strategy (DOTgEAr) *DOTgEAr* (Dataflow-Oriented Testcase-generation with Evolutionary Algorithms) was developed by Norbert Oster at the Inst. f. Information Processing, Friedrich-Alexander University, Erlangen-Nürnberg as PhD thesis.

DOTgEAr aims primarily at the generation of test cases for Java, but can principally also be used for other programming languages, even non-object-oriented, although the latter would not take benefit from the features especially developed for handling class instantiations, method invocations, and others. Of course, applying it to another language requires adaptation of the instrumentation and log-interpretation components.

The basic goal of DOTgEAr is to find minimal sets of test data for a given program (class) with maximal coverage according to a selectable number of control- as well as data-flow criteria. Since this implies at least two objectives to be considered at any time – namely minimisation of number of test cases and maximisation of coverage – the approach is intrinsically multi-objective. Further, since for all addressed coverage criteria the number of possible paths in general outgrows their exhaustive analyses, evolutionary techniques based on statistical methods have been chosen. Finally, in order to minimise maximum waiting time, a distributed execution of the test cases evaluation has been implemented.

From the control-flow oriented criteria, *branch coverage* and the *condition coverage* criteria

- simple condition coverage (SCC)
- condition/decision coverage (C/DC)
- minimal multiple condition coverage (MMDC)
- modified condition/decision coverage (MC/DC)
- multiple condition coverage (MDC)

are supported.

In addition data-flow oriented criteria are supported, which address paths from definitions of data to their uses. DOTgEAr knows about following definitions:

- *Control-flow graph G of a program P:* directed graph, where each node $N \in G$ represents either a branching statement (if, case etc.) of P or a sequence of unbranched statements, and each edge $E \in G$ a possible direct path from some N_i to N_j (i.e. without any other statements in-between).
- *Definition def($N_i, v_1, \cdots, v_k$):* set of all variables $v_1, \cdots, v_k$ for which values are assigned in the statements of N_i.
- *Computational use c-use(N_i, v_k):* given if in a non-branching node N_i the variable v_k is used in a reading mode.
- *Predicative use p-use(N_i, v_k):* given if in a branching node N_i the variable v_k is used in a reading mode.
- *Definition clear (def-clear):* a sub-path $(n_d, n_1, n_2, \cdots, n_m, n_c)$ of a control-flow graph G = (N,E) is def-clear with respect to variable v from n_d to n_c if $n_1 \cdots n_m$ do not contain a definition of v.
  Note: If n_d contains several statements using v, it should be split into nodes such that each new node contains mostly one reference to v.

Further definitions concern local/global use of variables, sets of variables defined or used in a node or vice-versa sets of nodes where some v is defined/used.

In addition, data-flow criteria according to [187] are defined:

- *all-defs:* for each variable v and each def(N_1, v) at least one def-clear sub-path to a node with use(N_2, v) is covered, with use = c-use or p-use.
- *all-c-uses:* for each variable v and each def(N_1, v) at each def-clear sub-path to a node with c-use(N_2, v) is covered.
- *all-p-uses:* for each variable v and each def(N_1, v) at each def-clear sub-path to a node with p-use(N_2, v) is covered.

To cover with possible empty sub-path sets for the last two criteria above, they can be extended to *all-c-uses/some-p-uses* and *all-p-uses/some-c-uses*, respectively. Further, *all-uses* integrates these criteria, but can lead to an infinite number of paths due to loops. This is weakened by *all-DU-paths*, which requires the subset of all loop-free paths of *all-uses*.

DOTgEAr allows to apply various meta-heuristics to generate and select test-cases, where currently following four are supported.

- *Random Search*

  1. An initial set of test cases is chosen by random, with a randomly chosen number of test data.
  2. Another set of test cases is chosen by random, with a randomly chosen number of test data.
  3. That test set (set of test cases) with higher fitness is kept.
  4. Steps 2 to 3 are repeated until a predefined number of iterations is processed or a maximum processing time consumed.

  The fitness is a weighted sum of number of covered paths according to the chosen criteria(s) and number of test cases, where the latter is of course 'inverted'. Windowing and normalisation operations can be configured, as the weights can be freely chosen.

- *Simulated Annealing*

  1. An initial test set is chosen by random, with a randomly chosen number of test data, and its fitness computed.

2. A new test set is derived from the currently hold by one (or several) of the following mutation operations: a.adding a new test case with randomly chosen test data (as long as a maximum number of test cases has not yet been reached) b.removing a test case (as long a minimum number of test cases has not yet been reached) c.modifying data of a test case
3. The new test set is applied, and its fitness computed.
4. If the new fitness is better than the previous, the old test set is replaced by the new one. Otherwise, the new test set is still taken with a probability

$$P(f_i, f_j, T) = \begin{cases} e^{-\frac{|f_i - f_j|}{T}} & \text{if } T > 0 \\ 0 & \text{otherwise} \end{cases}$$

where T is the 'temperature', which is decreased over the iterations.

Fitness and termination criteria are the same as for Random Search.

- *Multi-objective Aggregation (MOA)* In contrast to the previous two approaches, this is an evolutionary method. Therefore, not only one test set exists at a time, but a collection of test sets.

  1. An initial test collection is generated, consisting of a randomly chosen number of test sets, which are each generated as before.
  2. For each test set, its fitness is calculated as before.
  3. A predefined number of the best test sets is copied into the new generation (elitism).
  4. The missing test sets are derived from those in the current population by *selection* (of parents), *recombination* and *mutation* of test cases.

  Termination criteria are the same as before.

- *Nondominated Sorting Genetic Algorithm (NSGA)* Steps 1 and 4 are the same as for MOA, but the fitness of each test set is calculated using the *Pareto-Ranking*, where the number of test cases and the absolute coverage numbers according to the considered criteria are considered as individual dimensions. Consequently, those test sets are carried over into the next generation which are not dominated by any other test set (the *Pareto front*). It is possible to select the first k fronts of a generation (i.e. test collection), where the $(i+1)^{th}$ front is the set of non-dominated test sets after those belonging to front 1..i have been removed.

On test case execution, the source code for which test cases shall be generated first is exhaustively instrumented with logging statements. The instrumented code is the executed, and the log-files analysed to identify the paths covered according to the chosen coverage criteria(s). In particular, analyses of method invocations and hence paths crossing method borders is supported, taking pointer aliasing and multi-object instancing into consideration.

To reduce computing time, a distributed master/slave environment has been implemented, where test cases can be executed in parallel at a conceptually arbitrary number of nodes. In practice, up to 58 "remote execution engines" have been used.

The framework is completed by automatic generation test drivers (mock objects. etc.)

With 6 (fairly small) programs, comprehensible tests have been carried out, indicating that Simulated Annealing appears to be the heuristic which produces the best results, followed by NSGA and (more distant) MOA. Much more information can be found in [178].

An interesting aspect is combination of DOTgEAr with other tools. For instance, the 2 test cases for a program Hanoi detected with NSGA (yielding 100% branch coverage), killed approximately 75% of 227 generated mutants. 1000 more test cases, generated statistically, killed approximately 10% of the remaining mutants.

In summing up, benefits of DOTgEAr are

- Comprehensive work with a lot of reported test results
- Adaptable and highly configurable framework
- Support a wide variety of source code path coverage criteria

```
specification Sorted_List_Example : exit

library Boolean, Lists endlib

type SortedList is
    List, Elem, Boolean
opns
    sorted : List -> Boolean
eqns
    forall xs: List, a, b : Elem
    of sort Boolean
      sorted(nil) = true;
      sorted(cons(a, nil)) = true;
      sorted(cons(a, cons(b,xs)) = (a <= b) and
                                   sorted(cons(b,xs));
endtype
```

Figure 7: An algebraic specification of sorted lists as an extension of general lists.

As disadvantages can be seen

- Strong focus on Java source
- Invasive
- No measures for relative coverage (in general)

3.3 Testing from Abstract Data Types

The idea of testing code against an algebraic specification of an ADT is to show that the final system satisfies the axioms in the specification. To create tests for a given axiom, the variables of the axiom are instantiated with values. To run such a test, the resulting expressions are evaluated. If the results satisfy the axiom then the test is passed, otherwise it is failed.

The approach to generate test cases from ADTs was originally proposed by Gannon, Mc Mullin and Hamlet [95] and Bougé et al. [34, 35, 36], and then further developed and implemented by Bernot, Gaudel and Marre [24, 98]. It has been successfully applied to test procedural [18], functional [60] and object-oriented [78, 54] programs. Recently Gaudel and Le Gall published a survey on this technique [99].

Figure 7 shows an algebraic specification of sorted lists in the notation of (extended) LOTOS. Such a specification has two parts: a *signature* $\Sigma = (S, F)$, where S is a finite set of *sorts* and F is a finite name of *operation names* over the sorts in S, and a finite set of *axioms*. The sorts in our examples are `List`, `Elem` and `Boolean`; the new operation `sorted` is defined by three axioms.

An advantage of algebraic specifications in testing is the ease with which test data are constructed. The constructive nature of the specifications shows how to build various instances of the defined data types, which also implicitly include test sequencing information. There is also great potential for automatically generating test cases from specifications by steadily building up data from the base cases using constructor operations.

In the SortedList example, a possible test, inspired by the third equation, is:

$$sorted(cons(A, cons(B, cons(C, nil)))) = (A \leq B) \wedge sorted(cons(B, cons(C, nil)))$$

The corresponding test procedure for testing an implementation P consists of two computations and one comparison, namely

- computing the representation in P of $cons(A, cons(B, cons(C, nil)))$, calling the $sorted$ function in P on it, and storing the result;
- similarly, calling $sorted$ on $cons(B, cons(C, nil))$, checking if $A \leq B$ and building the conjunction of the results.
- comparing the two results (oracle).

This shows that in algebraic testing, the test-cases are equations and the job of an oracle is the comparison of the computed results of the left- and right-hand side of an equation.

The challenge of this method is to show that the equalities of the axioms hold, since this *oracle problem* is known to be undecidable, in general. Hence, (partial) reliable decision procedures are needed in order to compare values of terms computed by the IUT. Actually, implementations of abstract data types may have subtle or complex representations, and the interface of the concrete data types is not systematically equipped with an equality procedure (method) to compare values. In practice, only some basic data types provide a reliable decision procedure to compare values. They are said to be observable. The only way to define decision procedure for abstract data types is to observe them by applying some (composition of) functions yielding an observable result: they are called observable contexts. Observational approaches of algebraic specifications bring solutions to define an appropriate notion of correctness taking into account observability issues. [99]

3.4 Testing from Labelled Transition Systems

In case the system model is available in form of an LTS and the chosen conformance relation is ioco (see subsection 2.4.2), several different techniques for automated test case generation exist. These techniques mainly fall into the categories of *exhaustive testing*, *scenario-based testing* (TGV), *random testing*(TorX), and *fault-based testing*.

Before enumerating different strategies to generate test cases, we need to give the definition of a ioco test case [204] where S denotes the set of states, L_I is the input alphabet, L_U is the output alphabet, θ denotes quiescence, T is the transition relation, s_0 is the initial state, and $\emptyset$ is the empty set. The set $L = L_I \cup {}_U \cup \theta$ contains all observable actions.

Definition 9 (IOCO Test Case) *A test case t is a labelled transition system $\langle S, L_I \cup L_U \cup \{\theta\}, T, s_0 \rangle$ with $init(s) = \{a \mid s, s' \in S, a \in L : s \stackrel{a}{\Rightarrow} s'\}$ such that*

- *t is deterministic and has finite behaviour*
- *S contains the terminal states* **pass** *and* **fail**, *with $init($**pass**$) = init($**fail**$) = \emptyset$;*
- *for any state $t' \in S$ of the test case, $t' \neq$* **pass**, **fail**, *either $init(t') = \{a\}$ for some $a \in L_I$, or $init(t') = L_U \cup \{\theta\}$.*

The main property of the above definition is that a ioco test case is a tree with **pass** and **fail** verdicts on the leaves. A test case is required to be finite so running the test lasts for a finite time. It has to be deterministic, so the test executor has control over the testing process as much as possible: This means that a choice between input and output action, as well as a choice between multiple input actions are not allowed. (Input and output are seen from the perspective of the implementation under test.)

Running a test case is modelled by the synchronous parallel execution of the test case with the implementation under test. The test ends when a deadlock occurs. Due to the definition of the test case and the required *input enabledness* of the implementation, a deadlock can only occur in a **pass** or **fail** state. Of course, an implementation passes the test only, if it deadlocks in **pass**.

It is also mentioned in [204] that because an implementation may behave non-deterministically, different test runs of the same test case may lead to different verdicts.

This concludes the characterisation of an ioco test case. We now present strategies to select test cases using the input-output conformance relation.

Exhaustive Testing. A non-deterministic algorithm to build an exhaustive and sound test suite for ioco conformance tests can be found in [204]. Exhaustive test suites have the ability to assure that an implementation under test, when it passes all tests, is ioco conform to the specification. However, such a test suite may also reject ioco conforming implementations. A test suite is called sound, if it can detect non-conformance but can not assure conformance. The algorithm published in [204] is (theoretically) able to create a sound and exhaustive, thus complete, test suite. However, using the algorithm may not be feasible in practice.

Given a suspension automaton Γ_s of the specification, the basic observation behind the presented algorithm is that for ioco conformance it has to be checked for each $\sigma \in traces(\Gamma_s)$ whether the outputs of the implementation after σ are a subset of the allowed outputs, in other words, that $out(\Gamma_i \textbf{after} \sigma) \subseteq out(\Gamma_s \textbf{after } \sigma)$ holds. So whenever the implementation makes an output the test case has to check whether the output is allowed. A special feature of ioco is δ, quiescence, which models the absence of any output. Reading quiescence from the implementation matches θ-transition in the test case, which will go to the **pass** state if quiescence is allowed at that point, otherwise the θ-transition will lead to **fail**, if quiescence was not allowed.

Testing With Purposes. TGV [143], which is part of the *Construction and Analysis of Distributed Processes* (CADP, [96]) package, has been used in a number of case studies in different domains. Together with other testing tools, it is covered in [22].

TGV uses a supplied *test purpose*, for test generation. A test purpose thereby is a deterministic, complete IOLTS enriched with two kinds of trap states, Accept and Refuse and has the same alphabet as the specification. Complete means that the purpose allows all actions in each state (a special transition-label "*" is available). A trap state has loops on all actions, so it can not be left. The basic idea is to take a test purpose and calculate the synchronous product with the LTS of the specification. Thereby using the information about Acceptance and Refusal from the test purpose. After building the product, TGV adds special transitions representing quiescence. In a next step the result is determinised and test cases are extracted: All trace leading to an Accept state are describe valid behaviour.

Apart from creating single test cases, TGV can also be used to generate a test graph that describes all tests possible with a test purpose. A test graph might include traces that do not lead to either Accept or Refuse states. TGV truncates these traces and adds an Inconclusive verdict. Traces that reach Accept get a Pass verdict, while Fail verdicts are given implicitly for observations not present in the test graph. As all ioco tests have to be deterministic, TGV builds an controllable subgraph.

Relying on such a test graph, it is possible to use coverage criteria for test case selection [89]. Apart from using coverage criteria to specify which tests from all possible test cases of a certain scenario shall be selected, coverage criteria can also be used to select parts of the specification (e.g. decisions, actions) that need to be covered by tests. For example, in [88] the authors discuss action and decision coverage of LOTOS specifications.

STG [61, 195] builds on the ideas of TGV and TORX (see *Random Testing* paragraph) and adds symbolic test case generation on top. STG has been developed at IRISA/INRIA Rennes, France and uses a high-level, LOTOS like input language at the user level. The conformance relation in use is a symbolic form of ioconf without quiescence.

For test case generation, STG takes a specification, a test purpose and returns a symbolic test case that still has to be translated into a concrete test program. All, the specification, the test purpose, and the test case are input-output symbolic transition systems.

Additional properties of the test purpose are as follows: The test purpose can refer to parameters and variables of the specification, it does not have to be complete, and it includes special locations *Accept* and *Reject*.

Like in TGV, the test case is generated by computing the product of the specification and the test purpose. In order to obtain input-complete test cases, all internal actions are eliminated from the product. Next, by using an heuristic, any nondeterminism is removed from the product in order to avoid verdicts depending on the internal choice of the tester. In a last step, that part of the product is selected, that leads to the set of *Accept* locations, and the test case is made input-complete by adding transitions to the location *fail*. The result is an input-complete, initialised, deterministic, and correct test case.

In [144] the authors introduce a notion of quality for generated test cases and show how to deal with infinite-state symbolic models by applying approximate fixpoint analysis and frame the test selection problem as a game, with an

off-line and online part. The off-line part thereby can be seen as syntactic slicing of the specification with regard to particular scenarios. The online part relies on constraint solving for the choice of values of parameters.

During test case generation, STG uses OMEGA[5] to detect unsatisfiable guards and relies on NBac[6] for the approximate fixpoint analysis.

STG reports "Pass", "Fail", or "Inconclusive" as test results, depending on whether the test did not detect any errors, the test detected an error, or no errors were detected but the test purpose was not satisfied.

Apart from extending LTS with variables, leading to an STS , one can allow a recursive specification and then has to deal with a *push-down system* (PDS) . See [65] for details.

AGATHA [85, 26] is a tool for symbolic test case generation that relies on symbolic execution [97]. It has been used in various industrial contexts, e.g. testing the weapon navigation systems of Rafale and Mirage 2000-9 [160].

According to [26], AGATHA deals with specifications written in UML [193], SDL [141], STATEMATE [117], and ESTELLE [139]. All these input languages are translated into an IOSTS (sometimes referred to as EIOLTS) that is internally used. AGATHA, similar to STG, relies on OMEGA for constraint solving and uses a rewriting tool for expression simplification.

Test purposes are finite symbolic subtrees of the symbolic execution tree. While test purposes can be chosen by the user, the authors of [97] propose criteria to automatically compute test purposes. It is stated, that this is a response to industrial needs where engineers are not always able to define which behaviour they want to test.

AGATHA reports either PASS, FAIL, INCONC, or WeakPASS: PASS means that the SUT passed the test, FAIL means it failed the test, INCONC means conformance is achieved but the test purpose is not achieved, and WeakPASS means that the tool can not say for sure whether the test purpose has been achieved, due to e.g. a non-deterministic specification.

The main idea of symbolic execution of the IOSTS is *to replace concrete input values and initialisation values of attributes variables by symbolic ones with fresh variables and to compute the constraints on these values: the path conditions*. Symbolic execution of an IOSTS yields a symbolic execution tree whose set of paths denotes the set of all behaviours of the specification. Test purposes then select a finite set of paths in the symbolic execution tree. Using the ioco relation, AGATHA then generates tests for the SUT.

In [85] operators (composition, hiding) on the IOSTS are introduced in order to deal with component-based specifications.

Random Testing. TorX [207] was created in order to evaluate ioco. TorX employs a random-strategy to create test cases, can use test purposes to constrain selections, and uses an on-the-fly execution mode. Together with TGV, TorX has been described in [22].

TorX implements an on-the-fly mode which means that test execution and test case generation goes hand in hand: If one test-step has been executed, TorX further expands the system model to determine the next step. Note that this matches the idea presented for exhaustive ioco testing. Again, the major property of ioco is that for any trace of actions allowed by the specification, the implementation has to respond with an output that is allowed at that state. It is this property and the input enabledness of the implementation that makes on-the-fly test execution so easy.

Test case generation is similar to a random walk beginning from the initial state. If the specification contains nondeterminism, TorX follows multiple paths at the same time. TorX, according to the specification, switches between sending stimuli to the implementation and waiting for output of the implementation as the specification (and the random selection) suggests. This process is repeated as long as TorX has not reached an user-defined upper bound of steps and the outputs of the implementation match the specification (i.e. are allowed). In case an output is not allowed, TorX gives a fail verdict, if it runs out of steps it returns a pass verdict.

[5] http://www.cs.umd.edu/projects/omega/
[6] http://www.inrialpes.fr/pop-art/people/bjeannet/nbac/index.html

TorX is able to use test purposes, which basically are a set of traces over the visible labels (including quiescence). It uses them to constrain the walk: A test purpose may only allow a certain set of inputs/outputs at a certain state. As in TGV, constraining the walk might lead to situations, where the set of allowed actions in a state becomes empty. In this case, TorX *misses* the test purpose, which is similar to an inconclusive verdict. If TorX manages to reach the end of one of the traces of the test purpose, it reports a *hit*. So when a test purpose is used, TorX returns a tuple from $\{hit, miss\} \times \{pass, fail\}$ as test verdict. In [32] and extension for timed testing with TorX is given.

Fault-Based Testing Further work includes fault based test case generation: The idea is to mutate the original specification, extract a discriminating sequence of the LTSs representing the original specification and the mutated specification, use this discriminating sequence as test purpose and generate a test case by, e.g. calling TGV. This approach has been pursued in [5] with a case study and further improvements given in [6]. While the presented work relies on bisimulation to generate the discriminating sequence, [7] improves upon this by developing the theory of a fault and using a ioco-checker tool [216] to extract the discriminating sequence.

Queued Testing. The work in [182] proposes an approach to test input/output transition systems (IOTS) with queues. The approach requires deterministic specifications and provides input-enabledness of test cases so that the tester does not need to choose between inputs and outputs. The testing process is separated into two tasks, one generates input sequences and the other one observes according output sequences. The two tasks maintain queues where the IUT consumes inputs from the input process queue and stores according outputs in the output process queue. When the observer process encounters an empty output queue, recognised as quiescence, it judges about the system's behaviour. A system state where the implementation produces no outputs is referred to as *stable*. A trace which transfers the system into a stable state is called *quiescent trace*.

3.5 Testing with Model Checkers

Counter examples are interpreted as test cases with following definition:

Definition 10 (Kripke Test Case) *A test case* $t := \langle s_0, ...s_n \rangle$ *related to Kripke structure* K *is a finite sequence such that* $\forall\ 0 \leq i < n : (s_i, s_{i+1}) \in T$ *for* K.

The result of the test case generation is a *test suite* (or *test set*). A test suite TS is a finite set of test cases.

Definition 11 (Kripke Test Suite) *A test suite* TS *is a finite set of* n *test cases. The* size *of* TS *is* n. *The overall* length *of a test suite* TS *is the sum of the lengths of its test cases* t*:* $length(TS) = \sum_{t \in TS} length(t)$.

Test cases comprise a finite number of states n and can contain a loopback state forming a lasso shaped sequence. Such test cases have to unfolded according to [202]. The Kripke structure contains propositions over variables and the labelling function valuates each variable to according states. The set of variables are partitioned into input, output, and internal variables. A $module$ is defined by a Kripke structure with a set of input and output variables. During test case execution the tester provides inputs to the implementation and observes its outputs. It is assumed that the SUT behaves synchronously to provided inputs, i.e., it delivers the outputs immediately before the tester applies new inputs in the next state of the test case. If the outputs of the SUT match the specified outputs in all states of the test case we get an *pass* verdict, *fail* otherwise.

The following small example of a car controller (CC) from [180] illustrates the approach. The model FSM comprises three states, see figure 8. The input actions are *break* and *accelerate* and the output action is the resulting velocity. Listing 9 depicts the SMV specification of the model. When the user accelerates the car it gains speed from stop to slow to fast. On a break event the car immediately stops. If there is no acceleration the car will reduce speed.

In addition to the model, several simple requirement properties are expressed in LTL:

1. Whenever the brake is activated, movement has to stop.

$$\square(\texttt{brake} \rightarrow \bigcirc\, \texttt{velocity} = \texttt{stop}) \tag{13}$$

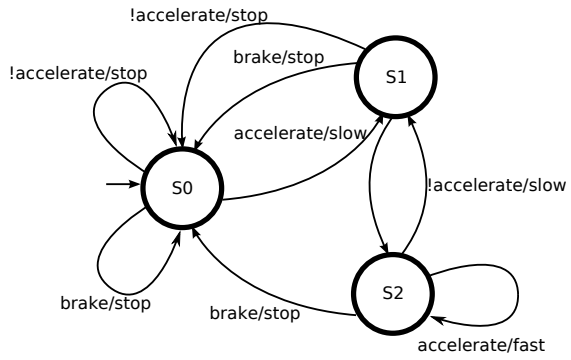


Figure 8: FSM of example model Car Controller (CC).

```
MODULE main
VAR
  accelerate: boolean;
  brake: boolean;
  velocity: { stop, slow, fast };

ASSIGN
  init(velocity) := stop;
  next(velocity) := case
     accelerate  & !brake & velocity = stop : slow;
     accelerate  & !brake & velocity = slow : fast;

     !accelerate  & !brake & velocity = fast : slow;
     !accelerate  & !brake & velocity = slow : stop;

     brake: stop;

     TRUE : velocity;
  esac;
```

Figure 9: CC represented as SMV model.

2. When accelerating and not braking, the velocity has to increase gradually, until it is fast.

$$\square(\neg\texttt{brake} \wedge \texttt{accelerate} \wedge \texttt{velocity} = \texttt{stop} \rightarrow \bigcirc\, \texttt{velocity} = \texttt{slow}) \quad (14)$$
$$\square(\neg\texttt{brake} \wedge \texttt{accelerate} \wedge \texttt{velocity} = \texttt{slow} \rightarrow \bigcirc\, \texttt{velocity} = \texttt{fast}) \quad (15)$$

3. When not accelerating and not braking, the velocity has to decrease gradually, until the car stops.

$$\square(\neg\texttt{brake} \wedge \neg\texttt{accelerate} \wedge \texttt{velocity} = \texttt{fast} \rightarrow \bigcirc\, \texttt{velocity} = \texttt{slow}) \quad (16)$$
$$\square(\neg\texttt{brake} \wedge \neg\texttt{accelerate} \wedge \texttt{velocity} = \texttt{slow} \rightarrow \bigcirc\, \texttt{velocity} = \texttt{stop}) \quad (17)$$

Coverage Based Test Case Generation It can be difficult to formulate "good" test purposes that achieve useful coverage on the specification or implementation respectively. Coverage criteria provide a means to cover the whole specification due to certain properties, e.g., generate a test suite that covers all states or all transitions. For each item to be covered a distinct *never-claim* property is formulated, called *trap property*. The set of counter examples generated by the model checker builds a test suite due to the formulated coverage criterion. For instance, if the n states of a specification are identified by an integer variable id we can achieve state coverage by formulating a set of following trap properties:

$$\square\neg(id = num) \text{ where } num \in 0, ..., n.$$

FShell [131] is an environment for white box testing of C programs. The basic approach is using model checking to find inputs that meet certain constraints such as structural constraints of the C program (in terms of function headers, labels etc.) as well as quality constraints like certain coverage criteria. The tool furthermore provides an interface with commands to define and manage test jobs in terms of primitives as well as for execution of the test jobs.

Model checking is done using the CBMC bounded model checker, which was extended to allow test case generation according to the above-mentioned constraints. CBMC uses a SAT solver as back-end to analyse the model. FShell extends this model by analysing the control flow graph and adding clauses to the generated SAT formula that describe the intended path. Furthermore continuously assumptions are added to the CNF to find the whole set of test cases corresponding to the coverage criteria with a single run of the tool.

Mutation Testing The idea of mutation testing is to introduce faults in either the specification or implementation and then generate test cases that discriminate specified from mutated behaviour. Mutation testing is based on the *coupling effect* [75] and the *competent programmer hypothesis* [4]. The first states that test cases which can detect simple faults are likely to find more complex faults. The latter states that programs are mostly correct. Programmers often make common mistakes like misnamed variables or wrong conditions in branch statements. Mutation testing uses these assumptions to form fault models comprising a set of mutation operators. Mutation operators for specifications are analysed by Black et al. [27]. The examples in [27] are modelled in SMV [153], the language for the SMV model checker.

Originally mutation testing considered faults introduced in the implementation [75, 4]. Specification mutation was initially proposed by Amman et al. [11]. In this context mutation testing deals with introducing faults in the model or in the temporal requirement specification. In the first case every property violation of the model leads to a counter example. These counter examples are *negative* test cases as they contain behaviour that must not be implemented by the SUT. In the second case temporal properties are mutated and produce counter examples as *positive* test cases.

3.5.1 Testing Real-Time Systems using UPPAAL

UPPAAL provides a model checking tool suite for black-box conformance testing of real-time systems described in [124]. Systems are specified as compositions of concurrent timed automata and related via *Relativised input/output conformance* to implementations. This conformance relation is consistent with untimed input/output conformance relation *ioco* of Tretmans [206]. The tool suite provides off-line and online test case generation. The first case produces shorter test cases because during generation the complete specification is considered and so better

structural coverage can be achieved. In many cases this leads to state space explosion. Online test case generation avoids the state problem and additionally can deal with non-determinism because states are explored dynamically. Since cases are generated event-by-event gained knowledge during test case execution can solve non-determinism. A disadvantage of online testing is that test sequences may get quite long because of the randomly explored state space.

Testing real-time systems demand some challenging issues like that the correct result of an operation is not sufficient but must be available within correct time. Additionally test cases must be executed fast enough so that no time constraints of the SUT are violated. UPPAAL exploits an adapted notation of timed automata in order to analyse them efficiently. The framework is based on the semantics of timed input/output transition systems (TIOTSs) . They extend the output actions of untimed input/output transitions with a set of clock variables $d \in \mathbb{R}_{\geq 0}$. Timed automata can be interpreted as extended finite state machines with a set of real-values clocks that can guard when transitions are allowed. Traces through the system contain beside input and output actions time information about clocks. The authors show in [124] the model of a light controller. Depending on the input timing of a user pressing a button the controller behaves differently. The user and the system behaviour are modelled with separate timed automata which are composed in parallel with matching input and output actions. It is assumed that the the system and the environment are weak input enabled, i.e., both of them cannot reject inputs with possible τ transitions in any state. A system S is non-blocking if for any state $s \in S$ and for any $t \in \mathbb{R}_{\geq 0}$ there is a timed output trace $\sigma = d_1 o_1 ... o_n d_{n+1}, o_i \in A_{out}$ such that $\sum_i d_i \geq t$. This means that the progress of time on the system side is never blocked by the environment and vice versa.

The authors of [124] introduce the definition of *Relativised Timed Conformance* by assuming non-blocking and weak input enabled systems/environments. The conformance relation is similar to *ioco* and considers traces in the environment to discriminate the output behavior of two TIOTS S and T composed in parallel with their environment E. For two states $s \in S, t \in T$ and $e \in E$:

$$s \ rtioco_e \ t \quad \text{iff} \quad \forall \sigma \in TTr(e) \bullet Out((s,e) \ After \ \sigma) \subseteq Out((t,e) \ After \ \sigma),$$

where $TTr(e)$ denotes the set of all timed traces in the environment, (s, e) is a composite state, and Out and $After$ are defined as in 2.4.2.

The UPPAAL tool can edit, simulate and check properties on UPPAAL timed automata where safety, liveness, deadlock, and response properties can be stated. Three different kinds of diagnostic traces can be produced: *some trace* leading to a goal state, the *shortest trace* with a minimum number of transitions, and the *fastest trace* with the shortest accumulated time delay. For reachability analysis states are represented symbolically as tuples (l, D) where l is a control location and $D \in \mathbb{R}_{\geq 0}^{|X|}$ is bounded by state invariants over clock values. Hence symbolic states may represent an infinite set of concrete states. In a symbolic computation step an action transfers the automaton into a state with a different control location and updated clock values. In [77] it is described how the convex subset of D can be represented as *difference bounded matrix* for efficient manipulation with constraint-solving techniques [191]. The model checkers UPPAAL and KRONOS [71] implement these approaches.

Test case generation is formulated as model checking problem where the reachability of a certain state is analysed. The searched state is described by comparisons of model variables with integer constants. A test case is a diagnostic trace $(s_0, e_0) \xrightarrow{\gamma_0} (s_1, e_1) \xrightarrow{\gamma_1} \ldots (s_n, e_n)$ where $s_i \in S, e_i \in E$ γ_i denotes either time delays or actions. Actions are divided into synchronising actions that are visible to both systems or actions that belong to only one system. A test sequence is obtained by projecting the action sequence to the environment thus invisible actions are removed and adjacent delays are summed up. Test cases can be generated due to test purposes by stating boolean expressions over states where it may be necessary to introduce auxiliary variables. Another possibility is to restrict the environment automaton to match the test purpose. Additionally UPPAAL supports the generation of test suites that satisfy certain coverage criteria either by formulating reachability properties or using observer automata [30]. In [63] different kinds of data flow coverage criteria are discussed.

An observer automaton is used to cover all items regarding a coverage criterion on a model automaton. This is achieved by superposition of model and observer. Whenever the observer reaches a state matching an item it accepts the trace leading to that state. As observer automata are non-deterministic they observe a set of location in the model automaton. It is possible to parameterise observers, e.g., "visit all locations L". Observer cannot monitor time delay because their predicates are only defined over computation steps. The observer based test case generation algorithm presented in [124] returns the trace which satisfies most coverage items. The observer concept is implemented in *UPPAAL cover* [125, 126].

UPPAAL also supports online testing which is able to deal with non-deterministic specifications. Non-determinism can be caused by abstraction or if systems depend on external events like interrupts. For online testing the algorithm presented in [124] performs three basic actions:

- send an input to the IUT.
- wait for an output for some time.
- reset the implementation and restart the process.

During test case execution the set of states is updated as inputs are sent and outputs are received. If the current set becomes empty we observe that the current trace is invalid resulting in a *fail* verdict. The online testing algorithm is implemented in the tool UPPAAL-TRON [157]. UPPAAL-TRON provides an API for programming test adapters. An industrial case study for an electronic refrigerator controller can be found in [158]. TRON provides three modes of operation:

- Pure environmental emulation where all traces in the environmental model are performed while accepting all possible output actions.
- Monitoring observes the system interaction with its real environment in a passive manner.
- Online testing can be performed by a combination of the two modes above. One component emulates the environment while the second component observes the implementation behaviour.

For many systems *timed trace inclusion* is too strict because the implementation is not allowed to be faster than the specification. The work in [64, 174] discusses alternative relations that allow such behaviour.

3.6 Testing from (E)FSM, State Charts

Testing with Purposes. *Autolink* [155] is an industrial strength test generation tool that generates test suites based expressed with the TTCN [196] language from SDL and MSC specifications. It has been used by the ETSI (European Telecommunications Standards Institute) to develop a test suite for Core INAP CS–2 [104].

MSC diagrams specify the interaction of the SUT with the test equipment. Based on these descriptions and the SDL specification, Autolink generates TTCN code that requires only little manual post processing. Autolink is integrated into the Tau tool set.

Test case generation is based on state exploration techniques and test purposes, i.e. paths: *A path is a sequence of events which have to be performed in order to go from a start to an end state in the state space of the SDL specification. The externally visible events of a path describe the test sequence to which a TTCN pass verdict is assigned.* Paths are represented as MSC diagrams and can be manually drawn, or taken from a simulation run of the SDL specification. They can also be generated automatically by some state space exploration algorithm that does an exhaustive and random walk and tries to maximise symbol coverage of the SDL system. A given MSC is valid (with respect to a given SDL specification), if a path in the state space of the SDL complies to the MSC.

A MSC consists of the SDL-specified system and a number of "PCOs" (Points of Control and Observation). Each PCO, that has its own time line, can send and receive events to/from the system. Given a MSC with more than one PCO (e.g., PCO1, PCO2) that all interact with the system: The interleaving of the events coming from PCO1 and PCO2 are not specified within the MSC! So, for example, if PCO1 sends two events and PCO2 sends two events, there are six different ways how these events can interleave. Of course, the sequence of events coming from PCO1 will stay the same (as specified), so if the MSC specifies that PCO1 sends first 'a' and then 'b' to the system, this ordering will be preserved. To clarify this further, supposed PCO2 wants to send a 'c' to the system, then it can send it either before the 'a' or before the 'b' oder after the 'b' of PCO1.

Given the above property, one MSC specifies a set of possible event sequences, so for test case generation, this set is checked against the SDL specification and TTCN test cases are generated for all sequences that are also traces of the SDL specification.

Given a MSC, Autolink can directly convert it to TTCN. This feature means that Autolink is able to create tests without specification – which has some advantages over writing TTCN test cases manually (see [155]).

A comparison of Autolink, TorX, and TGV can be found in [102].

Siemens Corporate Research built a test generation tool for UML models based on their "Test Development Environment" (TDE) [179], named *TDE/UML* [121, 120] . The tool has been recently enhanced by the CERTI Foundation to support UML 2.0 and to work as plug-in for the Eclipse platform [210]. It assumes that the implementation under test behaves deterministically and is externally controllable. For these implementations TDE/UML generates a set of conformance tests.

TDE/UML takes as input behaviour UML models annotated by the user with additional test data like coverage requirements and constraints, and produces a so called *test design* expressed in the "Test Specification Language" (TSL) . This language is part of the *category-partition method* [16]; it allows to define properties of the system's input and output in terms of *categories*, properties of individual input parameters that may distinguish the behaviour of the program (e.g., the length `l` of a string), *choices* defining ranges over each category (e.g., `l=0`, `l<=12` and `l>12`), and *rules*, that describe combinations of inputs that cause each result to occur. On the basis of these descriptions, the method determines behaviourally equivalent classes from which test cases are generated.

The TDE tool only generates test cases (for Java, C++) but cannot be used for executing them. For this stage a further tool needs to be used; the authors refer to JUnit as an example.

Random Testing *Guitar* [114, 166, 218] is a test case generation tool for Java GUI applications that uses AI planning [194] techniques and graph coverage criteria to automatically generate tests for an application.

The Guitar project comprises a set of components: A GUI ripper and an EFG generator that are used to automatically extract an event-flow model from an application, a test case generator that uses the EFG model to generate tests, a coverage evaluator that is able to calculate statement, branch, and path coverage, and lastly, jfcUnit that is used for test execution.

Test generation can be done in two modes: Either by random target-node selection in the EFG, or by random edge selection. Guitar then employs algorithms known from the field of AI-planning in oder to select operations that reach the selected node/edge.

Markov Chains *MaTeLo* (Markov Test Logic) [79] is a test case generation tool developed by All4Tec genie Logiciel. Based on the results of the EU FP5 project MaTeLo (project # 32402), it uses a combination of statistical usage testing and specification-based testing. For statistical testing, the concept of so-called MCUMs (Markov Chain Usage Models) is used. An MCUM is essentially a finite state machine with transition probabilities. MCUM can either be generated manually, or derived from requirements given as sequence charts (MSC – Message Sequence Charts, ITU standard Z.120 1996, or UML) or in SDL (CCITT standard). Here, also the input data for the test cases are derived from. Expected results can also be generated, using transfer functions defined with scilab/scicos© or Matlab/Simulink©. Such data are treated as input data to (and output data from) MCUM states.

Transition probabilities are initialised assuming equal distribution: For example, if after a certain "place" or "point" X a signal s is transmitted before some place Y in sequence chart C1 and a signal t before place Z in chart C2, then the MCUM contains a state X with transition s to Y and t to Z with probability 0.5 for each of them. These probabilities can be adapted by the user; in particular, several probability sets (called profiles) can be used. For modelling real-time aspects, self-transitions (from state X to X) can be used with respective probabilities.

Test cases are then generated considering the transition probabilities, i.e. test data are chosen according to them. Consequently, the more test cases are generated, the more unlikely paths through the MCUM will be covered. Generated test cases are currently represented in TTCN-3 (Testing and Test Control Notation version 3, ETSI standard ES 201 873-1) or TestStand format.

Benefits of MaTeLo are that it is fairly simple to use, has a large number of reference installations, the increasing trust with increasing number of test cases, the black box approach (good if system interior is unknown), and that it is not invasive.

Disadvantages are that the transition probabilities are needed, that no formal coverage measure is possible, that optimisation seems difficult, and that it is expensive as it is a commercial tool now.

3.7 Testing from Data-flow Models

Coverage Based Test Case Generation T-Vec Tester for Simulink [29] uses a test data selection algorithm that tries to find test-inputs near defined program input and output domain boundaries. The following model constraints are used to generate test vectors:

- Structured Path Coverage
- Decision (Branch) Coverage
- Modified Condition / Decision Coverage
- Statement Coverage
- Interface Coverage

T-VEC states that *the T-VEC Tester for Simulink automates much of the testing process by analysing the Simulink model to determine the best test cases for validating the model and testing implementations of the model. When used with the Real-Time Workshop™ Generic and Embedded Coders, the T-VEC Tester generates test drivers (harnesses) for executing the test vectors against auto-generated source code.*

In 2002, T-VEC has been used to identify a fault in the Mars Polar Lander [28].

Simulink®Design Verifier™ is a tool developed by The Mathworks, Inc that relies on the theorem prover of Prover Technologies and provides automated test case generation. Information about the tool is sparse, so we quote from the Mathworks' homepage: *Simulink Design Verifier generates tests for your Simulink and Stateflow models that satisfy model coverage and user-defined objectives. It also proves model properties and generates examples of violations. Simulink Design Verifier supports the following model coverage objectives: decision, condition, and modified condition/decision coverage (MC/DC). You can define custom test objectives directly in your Simulink or Stateflow models by using design verification blocks. With property proving, you can explore your design for flaws, missed requirements, and unwanted states, issues that are difficult to uncover by simulation alone.* The tool supports the following features:

- Generates tests for Simulink and Stateflow models
- Detects unreachable design elements in models
- Proves model properties and generates examples of violations
- Includes blocks for defining properties
- Produces test-generation and property-proving analysis reports

The *BEACON*[7] *for Simulink / Stateflow* tool from Applied Dynamics International (ADI) creates tests based on different path- and value related coverage criteria. In particular the tool generates input- and (expected) output vectors, a coverage summary report, and reports for selected analysis types. The homepage states that *because testing follows implementation, BEACON Tester generates the structural unit-level test cases needed to validate and verify the design. The complete structural test requirements dictated by the software design are accumulated, and test cases to cover them are formulated, all funnelled into a self-documenting test file ready for importation into your favourite unit test harness.*

[7] http://www.adi.com/products_be_bss_te.htm

Multi Paradigm *Reactis* [188, 200] automates the generation of test data for Matlab Simulink / Stateflow models. Test cases can be executed against the source code to evaluate implementation conformance or against the model to study the model's behaviour. Beside randomly generated input data *Reactis* provides an approach called *Guided Simulation*. The idea is to generate input values according to heuristics that ensure coverage on certain model properties.

AutoFocus [45] is a tool for developing distributed systems based on the semantics of communicating EFSM. The guards and assignments in the EFSMs are represented as functional programs. Typed channels between EFSMs enable a time synchronous communication.

The test case generator of *AutoFocus* takes a model of the SUT and a test case specification as input where the test case specification might be functional, structural, or stochastic. Test cases are automatically generated using Constraint Logic Programming (CLP) . In order to get executable test cases abstract and concrete data have to be mapped accordingly.

For test case generation the system model and the test case specification are translated into constraint logic programs (CLPs). The execution of the CLP yields the set of all possible execution traces.

3.8 Testing from Hybrid System Models

The authors of [148] propose a test case generation algorithm for hybrid systems and strategies to ensure coverage.

In [209] Michiel van Osch presents an test case generation algorithm based on the hioco conformance relation. The algorithm is not directly implementable and needs adaptions to be ready for practical use. For instance certain abstractions may reduce the infinite set of states and actions to countable sets.

The work in [15] presents an automated test generation approach for hybrid systems with discrete time. As specification formalism *Time Discrete Input-Output Hybrid Systems (TDIOHS)* are used. They build symbolic test cases and in a second step refine them with constraint solving techniques to executable test cases. The authors applied their tool in the embedded systems domain for testing avionics and the railway systems.

In [217] Franz Wotawa proposed to use Qualitative Reasoning (QR) for test case generation in the embedded systems domain. The work of Brandl et al. [40] uses *Garp3* [44] as QR modelling tool and deals with qualitative abstraction and modeling of ODEs. Furthermore they present a first idea of a test case generation algorithm based on reachability analysis of target states that are selected by test purposes. In [39] the authors present an approach of how to generate controllable test cases from QR models. Their tool *QRPathfinder* is still under development and comprises a test adapter interface to Matlab Simulink models.

The fault injection section has been kindly provided by SP; uses the fi.bib file for bibliography

4 Fault Injection Techniques

As fault injection (also known as fault insertion testing) has become widely used as an experimental dependability validation method, many different techniques for injecting faults have been developed. Fault injection accelerates the occurrences of faults in a system and the main purpose is to evaluate and debug error handling mechanisms. It is used at various abstraction levels and phases of the development process. Fault injection is e.g. mandatory in safety standard IEC 61508 (adapted by the automotive industry as ISO WD 26262) when claimed diagnosis coverage is at least 90%.

Fault injection is traditionally used for emulating hardware faults, where different techniques normally are divided into *simulation-based* and *physical* techniques depending on whether faults are injected into hardware models (e.g. VHDL models), or into an actual physical system or prototype. To avoid focusing only on the target for fault injection, the classification presented in the following sub-sections is instead based on how fault injection mechanisms are implemented. Thus, the focus is on whether extra hardware are used for fault injection (denoted as *hardware-implemented fault injection*) or if extra software is used (*software-implemented fault injection*) or if fault injection mechanisms are added directly into models of hardware, models of software or models of systems (*model-implemented fault injection*).

The various fault injection techniques can be characterised according to different properties. One such property is *reachability*, expressing the ability of the fault injection technique to reach possible fault locations in the system. Another property is *controllability*, with respect to *space* and *time*, denoting the ability to control *where* and *when* the faults are injected among the reachable locations. *Repeatability* denotes the ability to accurately repeat a single fault injection experiment while *reproducibility* refers to the ability to statistically reproduce the results of several experiments for a given set-up. *Intrusiveness* relates to the level of undesired impact the fault injection technique may have on the behaviour of the target system and can be divided into *space* and *time* properties. In order to achieve experiments corresponding to faults in the real world, it is important that the intrusion is low. Intrusiveness in time relates to the temporal overhead caused by the fault injection technique while intrusiveness in space relates to the hardware/software overhead. Other properties include *flexibility*, denoting the ease of changing fault injection targets in the system, *effectiveness* with respect to the ability to activate and exercise various fault handling mechanisms in the system and *efficiency* with respect to the amount of time and effort needed to conduct the experiments. Another important property is *observability*, which refers to the ability to provide means for observing and measuring the effects of faults in the system.

4.1 Hardware-Implemented Fault Injection

Hardware-implemented fault injection techniques (also referred to as "Hardware-based") are applied on actual implementations or prototypes of systems during later phases in the development process. While the observability and controllability can be limited the efficiency is often high. Another advantage is that the actual implementation of the system is validated instead of a system model.

One of the most common hardware-implemented techniques is *pin-level fault injection*. With pin-level fault injection [12], the faults are either injected by *forcing* the pins of ICs to faulty values using probes, or by *insertion*. The insertion technique uses a hardware fault injection module that is inserted between the pins of the fault injected target IC and the rest of the system. This allows the fault injected IC to be properly isolated from the other parts of the system.

MESSALINE [12] is a pin-level fault injection tool which uses forcing while the RIFLE [161] tool uses the insertion technique to inject faults. In RIFLE, only faults resulting in *effective errors* are considered, i.e. errors that are neither latent nor overwritten. This is achieved by comparing the results with a fault-free reference run. After the fault is injected, the faulty run is compared at fixed time intervals with the reference run. If the results are equal, the experiment is terminated since it will not result in an effective error. The appropriate time interval used will depend on the application. A fault is injected at a specific point during program execution when a certain bit pattern occurs at the pins of the IC, allowing reproducible faults to be injected. External faults are emulated by injecting a fault on the pins during the processor's read cycle. By injecting a fault on a suitable pin in the write cycle, an error can be produced at the addressed memory location.

The main drawback of the pin-level fault injection technique is the high intrusiveness on the system. Another

drawback is the limited reachability, since only the pins of the ICs are accessible and not the internal parts of the circuits.

4.1.1 Heavy-Ion and EMI Fault Injection

Heavy-ion fault injection [151] is an example of radiation induced fault injection. In the heavy-ion technique, the target circuits are bombarded with radioactive particles (heavy ions) generated by cyclotrons or emitted by Cf-252 sources. If the energy of the particles is sufficiently high, they may cause bit-flips in internal locations of the circuits. Another physical technique is electromagnetic interference (EMI) fault injection [150] which uses electromagnetic interference generated by a burst generator to inject faults. This is done by placing probes or plates connected to the burst generator above the target circuits.

The advantage of using heavy-ion fault injection or EMI fault injection is that they feature low intrusiveness on the system as compared to pin-level fault injection. Physical contact between the tool and the target system is lower or not even needed. Another benefit is the high reachability since all locations within ICs can be affected. One drawback of these techniques is that the results of single fault injection experiments are not reproducible. The results may be statistically reproducible however, i.e. the statistical measures are reproducible if a sufficient number of experiments have been carried out.

Three physical fault injection techniques have been investigated in an evaluation of the distributed fault tolerant architecture MARS [150]. The three techniques were pin-level fault injection, heavy-ion fault injection and EMI fault injection. Each fault injection technique gave different results, suggesting that they are complementary. Heavy-ion fault injection stressed the system the most causing the highest percentage of value failures to occur. Pin-level fault injection triggered more EDMs outside the CPU chip than the other techniques. This study also showed that end-to-end mechanisms (checksums and double execution of tasks) were required to increase the fault tolerance of the system.

4.1.2 Scan-Chain Implemented Fault Injection

Scan-chain implemented fault injection (SCIFI) uses scan-chains, i.e. built-in logic common in modern microprocessors for testing and on-chip debugging, to inject faults. A scan-chain can be compared to a shift-register connected to many of the internal state elements of the CPU (registers and cache) or to the pins of the CPU. Fault injection is conducted by reading out the shift-register, manipulating the contents, and then writing back the corrupted data.

The FIMBUL tool [86] injects faults using scan-chains conforming to the IEEE 1149.1 standard [175] and has been used to inject faults via the TAP[8] of the Thor CPU [1]. Transient faults are injected in locations accessible by *internal* and *boundary* scan-chains. The internal scan-chain is connected to many of the internal state elements in the CPU, e.g. registers and cache. The boundary scan-chain is connected to the pins of the CPU. A *debug* scan-chain is used to trigger the points in time for fault injection by allowing break-points to be set. In [86], the MEFISTO-C tool was used on a VHDL model of the Thor CPU while FIMBUL was used for injecting faults via the scan-chains of Thor, allowing a direct comparison between simulation-based (see Section 1.3) and scan-chain implemented fault injection. The comparison showed that the SCIFI technique can be more than 100 times faster than simulation-based fault injection and still produce similar results.

Other types of on-chip debug techniques exist beside those based on IEEE 1149.1. Background Debug Mode (BDM) is a proprietary technique from Freescale which has been successfully exploited for fault injection in [189]. Both SCIFI and BDM fault injection feature low space intrusiveness since built-in test logic is used for injecting the faults without any extra hardware being required. However, the BDM technique is limited to Freescale devices through the use of the proprietary BDM port. Other on-chip debugging techniques, e.g., those supporting real-time tracing such as the IBM RISCTrace and the Nexus standard [136] are also suitable for fault injection as they feature minimal time overhead for injecting faults and observing the system as the system does not need to be halted. Nexus-based fault injection experiments have been conducted with the GOOFI tool [212]. Another Nexus-based fault injection tool is presented in [219].

[8] Test Access Port

4.2 Software-Implemented Fault Injection

Software-Implemented Fault Injection (SWIFI) can be used to emulate the effects of physical faults with software in a physical system. SWIFI can emulate faults in various parts of the hardware such as CPU registers, the ALU and the main memory. A common technique is to generate an interrupt when faults shall be injected. The point in time at which faults are injected can be determined by *spatial* or *temporal* triggers. A spatial trigger is when a particular address is accessed during execution. A temporal trigger is a pre-determined time from the start of the execution of program. When the interrupt is generated, an interrupt handler is executed to manipulate, for example, the contents of CPU registers or the operation codes or operands of machine instructions. The various SWIFI techniques can be divided into *pre-runtime* injection techniques and *runtime* injection techniques depending on whether the faults are injected before the system starts executing the software or during the software execution.

4.2.1 Pre-Runtime SWIFI

With pre-runtime SWIFI, one or more bits are inverted in the program and/or data image (i.e. in the executable file) before the program is executed, see e.g. DOCTOR [116]. There is no intrusion on the execution time using pre-runtime SWIFI, but the fault models are limited. Typical fault models are memory faults at the point in time the memory is first accessed.

4.2.2 Runtime SWIFI

In runtime SWIFI, faults are injected during the execution of the application. The Xception tool [49] uses advanced debugging and performance monitoring features that already exist in many modern processors, e.g. Pentium, Alpha, MIPS and PowerPC, to inject faults. In contrast to many other SWIFI tools, fault injection with Xception does not require the target application to be modified with, for example, trap instructions. It is necessary to only add fault injection interrupt handlers. To be able to inject faults into real-time systems at full speed without causing any deadlines to be missed, the maximum execution time for the fault injector must be minimised and predictable. For this purpose, an enhanced version of the Xception tool has been developed resulting in the real-time fault injection tool RT-Xception [69]. The RT-Xception tool features an upper bound on the maximum number of executed instructions in the fault injection code, which makes the execution time for each fault injector predictable and thus applicable for real-time systems.

Another tool that takes advantage of processor debugging facilities is MAFALDA [84]. The aim of this tool has been to evaluate the use of COTS[9] microkernels in safety-critical systems.

The FERRARI tool [149] uses UNIX operating system calls to carry out fault injection. Hardware faults and errors are emulated by corrupting the program state during execution. This results in the same system behaviour as would be found if an internal fault had been present, e.g. bit-flips in a CPU register. FERRARI uses three methods for fault injection, pre-runtime SWIFI, spatial triggers and temporal triggers. The spatial trigger method employs a user-selected address to trigger fault injection. The temporal trigger method employs a built-in timer count down register loaded with a user-determined value. When the register reaches zero, the processor is halted and a fault is injected.

Software-implemented fault injection has also been used for validating object-oriented applications during runtime. Here, the fault models are erroneous inputs and outputs from the target software and erroneous data within the target software. FIRE [163] and Jaca [164] uses reflective programming to inject and monitor C++ and Java applications respectively. The user of the tool decides whether the target software should be monitored, fault injected or both. Using a special pre-processor, additional objects are created, so called meta-level objects. The meta-level objects are compiled together with the target software and used to manipulate the original target software at runtime.

The PROPANE (PROPagation ANalysis Environment) tool [129] injects faults in C-code executing on desktop computers. The code is instrumented with fault injection mechanisms and probes for logging traces of data values and memory areas. PROPANE supports the injection of both software faults (by mutation of source code) and

[9] Commercial Off-The-Shelf

data errors (by manipulating variable and memory contents). Common error types are provided and user-defined error types are also supported.

4.3 Model-Implemented Fault Injection

Model-implemented fault injection is defined in this survey as a technique where fault injection mechanisms are inserted into:

- Hardware models, e.g. in VHDL specifications, at a development phase where no physical prototypes are available. Fault injection in hardware models are usually denoted as simulation-based fault injection.
- Software models where a fault may e.g. result in a data error.
- Models of system where a fault can e.g. emulate a faulty sensor, actuator or a sub system.

4.3.1 Fault Injection in Hardware Models

Simulation-based fault injection can be used to study effects of faults in computer systems at various levels of abstraction. Simulations can be conducted at the electrical circuit level, the gate level or at various subsystem and functional levels. Validation of software-implemented error detection and recovery mechanisms requires complex simulation models that can simulate the program execution. To ensure the accuracy of the results, these models must consist of accurate representations of the CPU, the main memory and I/O devices. The main advantage of simulation-based fault injection is that simulations can be carried out early in the design cycle, before a physical prototype of the system is available. Simulations also provide high observability and controllability. Observability refers to the ability to provide means for observing and measuring the effects of faults in the system and controllability refers to the ability to control *where* and *when* the faults are injected among the reachable locations The main drawback is that simulating the execution of a computer can be many orders of magnitude slower than the execution in a real system. A common way is to use simulation-based fault injection at the logic (gate) level by injecting faults into VHDL[10] models of systems.

The MEFISTO tool [145] uses two approaches for injecting faults into VHDL models. The first is based on the use of *saboteurs* and *mutants*. Saboteurs are extra VHDL components that carry out the fault injection, while mutants are modified (faulty) VHDL components replacing the original (fault-free) ones. The built-in commands of the simulator is used to set signals and variables of the target simulation model to incorrect values.

The VERIFY tool [199] uses an extension of VHDL. In the description of each basic component (e.g. AND, NOT and OR gates), the mean time between faults, their duration and their effects can be specified. These parameters can be adjusted for the environment in which the application will be used, e.g. for space or nuclear environments. This approach allows the hardware manufacturers that provide or use VHDL libraries to adjust and simulate their designs on the basis of field data on, for example, fault occurrences in their components.

DEPEND [103] is an integrated design and fault injection environment. DEPEND supplies a C++ library of elementary and complex objects such as fault injectors, CPUs, N-modular redundant processors, communication links, voters and memory. These objects can be used to build and analyse a wide range of fault-tolerant architectures. The system behaviour is described by means of processes interacting with each other, allowing software applications to be executed by software-implemented fault-tolerant hardware architectures at the function (algorithm) simulation level.

4.3.2 Fault Injection in System and Software Models

In the ESACS [66] and ISAAC [185] projects, a technique has been developed which inject faults in system models developed with the modelling language SCADE, to simulate failures of external hardware components, for example sensor failures. The technique exploits the principle of minimal cut sets (MCS) which is well-known from risk analysis methods like fault tree analysis (FTA) to identify combinations of faults causing a safety requirement

[10] Very high speed integrated circuit Hardware Description Language

violation. However, instead of using static fault trees, design models have been extended with faults to calculate MCS lists considering also cycles (time stamps) of the model execution.

Another model-implemented fault injection approach for models developed with SCADE is evaluated in [213]. Here, the authors present a tool which automatically replaces original operators with fault injection nodes (FINs) . A FIN is a node that encapsulates the original operator so the operator can be replaced or the operator output can be manipulated. During execution of the generated source code, FISCADE controls the SCADE simulator to execute the model, inject the fault, and log the results. The tool allows the user to inject faults in all signals in the model. Furthermore FISCADE can simulate design errors by automatically replacing one operator with another operator, as well as simulating transient, intermittent or permanent faults affecting memories and CPU registers. The tool automatically performs a pre-injection analysis to reduce the number of fault injection experiments needed. The tool also supports the work of configuring and carrying out automated fault injection campaigns.

Work has also been done to inject faults at a functional level via MATLAB/Simulink to evaluate automotive control functions [214]. Here, abstract blocks (for example implementing an adaptive cruise control or an active body control) are marked as faulty and the effects of other blocks are monitored during simulation.

Another work targeting fault injection at a functional level in Simulink models is presented in [137]. During execution of the model, fault injection blocks are added and executed to inject faults, e.g. bit-flip faults, stuck-at faults and faults resulting in an offset or amplification of a signal. The results from the experiments are compared with a fault-free simulation, i.e. a golden run. In the analysis phase, experiments which have an impact on the simulation can easily be collected by a graphical user interface (GUI) and statistical data can automatically be generated from the GUI.

In [147], an approach called *model based safety analysis* is suggested where executable models of the system are executed by using existing commercial tools such as Simulink and SCADE. To investigate how well faults are tolerated, the original model is extended with, for example, the stuck-at fault model and the extended model is formally verified using the SCADE Design Verifier.

4.4 Hybrid Fault Injection

Several techniques using combinations of some of the fault injection techniques mentioned above have been proposed. One example is given in [110] where SWIFI is combined with simulation based techniques which perform the actual fault injection. This *hybrid fault injection technique*, also known as mixed-mode fault injection, allows the advantages of both SWIFI and simulation based fault injection to be utilised, i.e. the actual target system may be executed at full speed except during the injection of a fault when a simulator providing detailed access to the target system is used instead.

5 Industrial Case-Studies

The preceding sections gave an overview of tools and techniques for model-based test case generation. Until yet we have not addressed the question of how relevant these techniques are for industrial sized settings. We address this shortcoming in this section by presenting case studies taken from industry.

We start with the GSM [23, 208] case study that use the B notation and the B-Testing-Tools (now cyalled Leirios Test Generator, LTG) to compute test cases. Next we discuss case studies of the AGEDIS [23] project that was *a European Commission sponsored project for the creation of a methodology and tools for automated model driven test generation and execution for distributed systems*. Finally, we look at formal methods in the area of railway systems.

5.1 GSM 11-11 Standard Case Study

This section presents the GSM 11-11 standard case study that was published in [23] and recently has been recapitulated in [208]. The GSM case study is interesting out of several reasons: It is well documented, it evaluates automated test generation in an industrial context, and the case study uses a mature hand-crafted, high-quality test-suite to evaluate the quality of the derived test cases.

Model The functions of the GSM 11-11 standard were formalised in B [2]. Note that the B–method allows to develop software by means of refinement, which – in the end – leads to a proven program. However, for test case generation, only the most abstract level of B specifications was used and the implementation under test was developed using a classical (design and coding) approach.

The basic construct of B is the abstract state machine that contains encapsulated data and operations to manipulate the data. It is in the spirit of earlier set-based specification languages, such as VDM and Z. (See section 2)

Figure 10 on page 46 sketches the derived model and gives a feeling for the B syntax. After modelling the GSM 11-11 standard in B, the model was verified using proof tools of *Atelier B*: At the abstract level, proof obligations for showing that the operations do not break the invariant are generated. It is reported that from the 377 obligations, 376 were automatically proved and 1 was satisfied using the interactive proof tool.

Test Generation Given the correct model, the B-Testing-Tool first does a boundary state computation. A boundary state is a state where at least one state variable has a value at an extremum of its subdomains. In order to compute the boundary goals[11] a partition analysis is performed. Boundary goals are obtained by searching for maximum and minimum solutions in the partition analysis of each operation: This is done by first translating the postcondition into DNF and conjoining it with the precondition and the invariant and then using a constraint solver. After determining boundary goals and instantiating them into boundary states, test cases are generated by traversing the constrained reachability graph to each boundary state of the specification. All paths leading to the boundary states are the initialisation parts of tests.

For each operation in every boundary state, BTT then does a boundary analysis of the input variables: Each operation is executed in a given boundary state as many times, as it has different possible input boundary values. Each of these executions represents one test. BTT also generates the oracle from the specification and adds a postamble for each test case in order to avoid manual initialisation after each test case.

Evaluation BTT identified 58 boundary goals and 42 different boundary states. From each boundary state, about 24 different invocations were performed, and about 1000 test cases were generated. (Calculation time was about two hours on a Sun Ultra Sparc 200 MHz CPU with 256MB RAM.)

[11] which is a subset of the system state, where at least one variable has an extremum; a boundary state may thus satisfy several boundary goals

Compared to the manually crafted test suite, it is reported that more than 85% of the manually generated tests were covered by the tests generated from BTT and 50% of the generated test cases were new ones. It is also reported that the test design time analysis revealed a time saving of 30%.

5.2 AGEDIS

The AGEDIS (Automated Generation and Execution of Test Suites in Distributed Component-based Software,[119, 118, 22]) project was a three year project running from November 2000 until February 2004. The language and tool development efforts were done by IBM Haifa Research Laboratory, Verimag, IRISA, Oxford University, Intrasoft International, and imbus AG. Experiments were carried out by Intrasoft International, France Telecom, and IBM UK.

The goal was the development of tools and a methodology for the automation of software testing with the emphasis on distributed component-based software. As specification language a UML-subset was chosen and the tools TGV (see 3.4) and GOTCHA (Generation Of Test Cases for Hardware Architectures, [90]) were used for automated test generation.

Most interesting from the final report [119] is the "lessons learnt" section. Decisions that turned out to be very positive were using UML as modelling language (because of industry acceptance), the re-use of existing tools, the XML format for the test suite, and the common test execution framework. More interesting, though, are points that would be done differently in hindsight:

- Instead of using IF [38] as action language it is noted that OCL or a subset of Java would have been a wiser decision. This is justified by customer needs and standards.
- Instead of using a free closed source UML modeller, which later wasn't available freely anymore, it is said that it would have been better to rely on an open source UML modeller of lower quality.
- It is also noted that XMI 2.0 should have been chosen as interface language.
- Generally it is remarked that it would have been positive to provide mode "instant gratification" to customers (e.g. by also using random test generation) and it seems that TGV and GOTCHA did not entirely meet the expectations.
- It is also noted that it would have been positive to get more and earlier feedback from users, providing information about what is relevant in industry.
- Finally it is reported that the GUI should have been integrated in an open source IDE, e.g. Eclipse.

During the project five case studies were carried out, three of them using the AGEDIS tools. The following is a quote from [119]: *A common feature of all the experiments was the fact that the AGEDIS methodology was never used as it was intended. In all cases, the system under test was already in existence when the experiments took place – thus we have no true test of the methodology in a production environment. However, the tools were tested and refined and in particular, the results of Experiment 4 are particularly encouraging.*

Case study 4 [67, 119] was carried out at Intrasoft with a first GUI version of the AGEDIS tools. The purpose was to compare the testing of a graphical web based application using the AGEDIS tools with the usual testing methods of the company. The case study was started in July 2003 and was finished in September 2003 without completing all aims of the study. Reasons given for not completing all aims are

- First use of the GUI version, which meant bug fixing
- First testing of graphical web application with AGEDIS
- Steep learning curve

In [119] the author gives a figure listing days needed for the testing process and compares AGEDIS with WinRunner, the tool normally used by the company. Out of seven categories where figures are listed for AGEDIS and WinRunner, AGEDIS needs less time than WinRunner in only one, namely "test cases creation". In category "execution", AGEDIS

needed 10 days while WinRunner needed one. Unfortunately, the results are not well commented, but the time difference can be due to the fact that AGEDIS created more test cases than the test engineer. Manual test case generation needed 5 days, test case generation with AGEDIS one day. Analysing the results needed 4 days for the AGEDIS generated tests, and two days for the WinRunner tests. Test plan creation, which only had to be done for WinRunner, needed one day. Simulation of the model, which only had to be done for the AGEDIS tools, needed five days. Also, recording test cases in WinRunner was quicker than modelling the system for AGEDIS.

Advantages when using a model-based testing procedure were found to be: The model could be used as a unifying point reference, that test results can be based on matches and deviations from what the model specifies and the software does, that a model provides a more effective way to analyse complex requirements, that changes to the requirements (model) are cheap because test cases can easily be regenerated, and that more tests (for better coverage) could be generated automatically than by the manual approach.

In [118] the authors sum up the experiences gained with the case studies by saying that the act of modelling was received well, while the industrial testers were critical of the modelling language and the test generator. *In particular the test generation algorithms proved not to be scalable to large industrial problems, and in each case the models were restricted to a subset of the SUT functionality, rather than testing the entire system.*[118]

5.3 Railway

Within the CENELEC (European Committee for Electrotechnical Standardisation) EN50128 standard that addresses the development of software for railway control and protection systems, using formal methods for the development of safety critical systems is rated as "highly recommended".

One very well known work in the area of formal methods and railway systems is the development of the safety critical parts of Line 14 of the Paris Métro [3] using the B-method of refinement.

Apart from model checking and formal proof, examples of railway system related work are: Automated test execution [51] using TTCN–3 [196], grey-box testing [172], as well as automatic model-based test case generation [31]. In [13] the authors report the story of the introduction of formal methods in the development process of a railway signalling manufacturer. Tools used in these examples range from SCADE, over STATEMATE to Stateflow.

Most relevant to this survey is the approach presented in [31]. The work is based on STATEMATE from I-Logix and uses *state-charts* and *live sequence charts* as system model. The tool described can use transition or state coverage as quality metric. Test case generation is based on *precondition computations for test goals and guided simulation. ... we compute in a backward manner conditions that have to be fulfilled immediately before the goal is reached. With these conditions, we iterate a precondition computation algorithm, label the design with conditions, and mark these with gradients that measure their relevance for or distance to the test goal. In a subsequent phase, a simulation of the design is guided by this precondition-based so-called gradient field. Test vectors, describing the used inputs as well as the observed outputs are stored during the simulation. Backtracking and dynamic modifications of the gradient field with respect to already examined states and transitions allow* the tool *to find the traces to/for the test goal.*[31]

Unfortunately, though, no real case study is presented in detail.

5.4 ASML EUV Machine

In [42] the authors propose a *model-based integration and testing* (MBI&T) method and evaluate it within an industrial context. In the following, we give a brief summary of the findings.

The MBI&T method of system development consists of following steps (taken from [42]):

1. Modelling the system components and the infrastructure based on design documentation
2. Model-based analysis of the *integrated* system model using techniques like simulation and model checking to analyse particular scenarios and system properties derived from the requirements and the design documents.
3. As soon as the realization of a component becomes available:

   (a) Model-based component testing of the component's realization (e.g. implementation) with respect to its model using automatic model-based testing techniques and tools

   (b) Replacement of the component's model by its realization

   (c) Model-based system testing of the integrated system (comprising e.g. component models and realization of components) obtained in the last step by executing test cases derived from requirements and design documents.

4. After all models have been substituted by realizations: testing of the complete system realization by executing test cases derived from the requirements and the design documents.

There already are tools in industry, e.g. Berner&Mattner's MODENA, dSpace, etc., that allow model-based simulation of system components. These tools can be used for integration testing with concrete realizations of system components. Note that not all components are always modelled in industry: Main reasons for not pursuing the model-based approach are that testing with the "real" part as "model" is sometimes cheaper, or the inability to create a model that matches the behaviour of the component. The main contribution of the paper is to set out a methodology that joins model based development with automated test generation and hardware / software simulation.

While the case study given in [42] focuses on all steps except automated model based testing, the article in [41] focuses on automated test generation. Consequently, we'll briefly discuss the techniques used in [41].

The case study given in [41] deals with a laser subsystem that is modelled in χ. Test case generation was done with Torx (see 3.4). TorX supports TROJKA as input language, which is a slightly modified version of PROMELA, the specification language for the SPIN model checker. Because SPIN is used within the χ tool set a translator of χ to PROMELA exists. Note that χ includes the notion of time that is not supported by TorX.

In the following we summarise the case study according to the steps of the MBI&T methodology.

1. The first step was to create models of the components. The system was modelled using χ but care was taken that the model could be translated to PROMELA (TROJKA). The resulting χ system contained 350 lines of code.

2. After creating the model, certain properties were verified using the SPIN model checker: Absence of deadlock, a system invariant concerning the translation of a specific χ statement, and model specific behavioural properties (e.g. order of state transitions, possible actions and responses)

3. Based on the verified (according to the given properties) model, a TROJKA model with about 1000 lines of code was generated. For translating the abstract commands given by TorX into "real" commands (and in reverse) an adapter component was written using PYTHON. Subsequently, TorX was used to carry out tests, which revealed bugs in the error handling code of the component.

Widening the focus again, leaving automated test case generation, the case study given in [42] highlights that often times the modelling step alone is worth the effort because it makes under-specified and/or conflicting requirements (or designs) explicit. In case of the ASML case study, pursuing the MBI&T method payed off, since expensive clean room time was saved, and no expensive fixing was necessary during real system integration. It is also noted that often times testing with models is easier because a model can be totally controlled as opposed to a real hardware system, where unforeseen "small" problems are likely to occur.

```
MACHINE GSM

SETS
   FILES = {mf,df_gsm,....};
   VALUE = {true, false};
   CODE  = {c1,c2,c3,c4}; ....
CONSTANTS
    MF == {mf};
    DF == {df_gsm};
    EF == {....};
PROPERTIES
    FILES_CHILDREN : (MF \/ DF) <-> FILES & ...
VARIABLES
    current_file, ...
INVARIANT
    current_file <: EF &...
INITIALIZATION
    current_file := {} || ....
OPERATIONS
    RESET =
       BEGIN
          current_file := {} ||
          current_directory := mf ||
          permission_session := {(always,true),(chv,false),
              (adm,false),(never,false)}
       END;

    sw <-- VERIFY_CHV(code)=
        PRE
            code: CODE
        THEN
            IF (blocked_chv_status = blocked)
            THEN
                ...
            ELSE
               IF (pin = code)
               THEN
                   ...
               ELSE
                   ...
               END
            END
        END
    END;
END;
```

Figure 10: GSM

List of Abbreviations

ADT ..... Abstract Data Type
AI ..... Artificial Intelligence
ALU ..... Arithmetic Logic Unit
API ..... Application Programming Interface
ATM ..... Automated Teller Machine
BDD ..... Binary Decision Diagram
BDM ..... Background Debug Mode
BNF ..... Backus-Naur Form
BTT ..... B Testing Tool
C/DC ..... Condition/Decision Coverage
CADP ..... Construction and Analysis of Distributed Processes
CCITT ..... Commité Consultatif Internationale Télégrafique et Téléfonique
CCS ..... Calculus of Communicating Systems
CENELEC ..... Comité Européen de Normalisation Electrotechnique
CIL ..... Common Intermediate Language
CLP ..... Constraint Logic Programming
CMBC ..... Ansi-C Bounded Model Checker
CNF ..... Conjunctive Normal Form
CPU ..... Central Processing Unit
CSP ..... Communicating Sequential Processes
CSP ..... Constraint Satisfaction Problem
CTG ..... Complete Test Graph
CTL ..... Computation Tree Logic
DART ..... Directed Automated Random Testing
DNF ..... Disjunctive Normal Form
DOTgEAr ..... Dataflow-Oriented Testcase-generation with Evolutionary Algorithms
EDM ..... Error Detection Mechanism
EFG ..... Event-Flow Graph
EFSM ..... Extended Finite State Machine
EIOLTS ..... Extended Input Output Labeled Transition System
ELOTOS ..... Extended LOTOS
EMI ..... Electromagnetic Interference
ETSI ..... European Telecommunications Standards Institute
FIN ..... Fault Injection Node
FSM ..... Finite State Machine
FTA ..... Fault Tree Analysis
GSM ..... Global System for Mobile communications: originally from Groupe Spécial Mobile
GUI ..... Graphical User Interface
HIOCO ..... Hybrid Input Output Conformance
HML ..... Hennessy-Milner Logic
IC ..... Integrated Circuit
IDE ..... Integrated Development Environment
IEC ..... International Electrotechnical Commission
IEEE ..... Institute of Electrical and Electronics Engineers
IOCO ..... Input Output Conformance
IOLTS ..... Input Output Labeled Transition System
IOSTS ..... Input Output Symbolic Transition System
IOTS ..... Input Output Transition System
ISO ..... International Organization for Standardization
ITU ..... International Telecommunication Union
IUT ..... Implementation Under Test
JML ..... Java Modeling Language
LOTOS ..... Language of Temporal Ordering Specifications
LTG ..... Leirios Test Generator

LTL ..... Linear Time Logic
LTS ..... Labelled Transition Systems
MBTCG ..... Model-Based Test Case Generation
MC/DC ..... Modified Condition/Decision Coverage
MCS ..... Minimal Cut Set
MCUM ..... Markov Chain Usage Model
MDC ..... Multiple Condition Coverage
MMDC ..... Minimal Multiple Condition Coverage
MOA ..... Multi-objective Aggregation
MOGENTES ..... Model–based Generation of Tests for Dependable Embedded Systems
MSC ..... Message Sequence Chart
NSGA ..... Nondominated Sorting Genetic Algorithm
NuSMV ..... New SMV
OCL ..... Object Constraint Language
ODE ..... Ordinary Differential Equation
PDS ..... Push-Down System
POSIX ..... Portable Operating System Interface
QDE ..... Qualitative Differential Equations
QR ..... Qualitative Reasoning
SAL ..... Symbolic Analysis Laboratory
SAT ..... Satisfiability
SCC ..... Simple Condition Coverage
SCIFI ..... Scan-Chain Implemented Fault Injection
SDL ..... Specification and Description Language
SIP ..... Session Initiation Protocol
SMV ..... Symbolic Model Verifier
STG ..... Symbolic Test Generator
STS ..... Symbolic Transition System
SUT ..... System Under Test
SWIFI ..... Software-Implemented Fault Injection
TAP ..... Test Access Port
TDE ..... Test Development Environment
TGV ..... Test Generation from transitions systems using Verification techniques
TIOTS ..... Timed Input Output Transition Systems
TSL ..... Test Specification Language
TTCN ..... Testing and Test Control Notation
UML ..... Unified Modeling Language
VDM ..... Vienna Development Method
VHDL ..... Very High Speed Integrated Circuit Hardware Description Language
XMI ..... XML Metadata Interchange
XML ..... Extensible Markup Language

References

[1] Saab Ericsson Space AB. Microprocessor Thor, product information. Technical report, 1993.

[2] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[3] Jean-Raymond Abrial. Formal methods: Theory becoming practice. *J. UCS*, 13(5):619–628, 2007.

[4] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report, School of Information and Computer Science, Georgia Inst. of Technology, Atlanta, Ga., Sept. 1979.

[5] Bernhard K. Aichernig and Carlo Corrales Delgado. From faults via test purposes to test cases: On the fault-based testing of concurrent systems. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, volume 3922 of *LNCS*, pages 324–338. Springer, 2006.

[6] Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer, and Franz Wotawa. Protocol conformance testing a SIP registrar: An industrial application of formal methods. In Mike Hinchey and Tiziana Margaria, editors, *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods*, pages 215–224, London, UK, 2007. IEEE.

[7] Bernhard K. Aichernig, Martin Weiglhofer, and Franz Wotawa. Improving fault-based conformance testing. *Electronic Notes in Theoretical Computer Science*, 2008. to appear.

[8] Rajcev Alur, Costas Courcoubetis, and David Dill. Model-Checking for Real-Time Systems. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS 90)*, pages 414–425, 1990.

[9] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular specification of hybrid systems in charon. In *HSCC '00: Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, pages 6–19, London, UK, 2000. Springer-Verlag.

[10] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998.

[11] Paul Ammann and Paul E. Black. A Specification-Based Coverage Metric to Evaluate Test Sets. In *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 239–248, Washington, DC, USA, 1999. IEEE Computer Society.

[12] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.

[13] Stefano Bacherini, Alessandro Fantechi, Matteo Tempestini, and Niccolò Zingoni. A story about formal methods adoption by a railway signaling manufacturer. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 179–189. Springer, 2006.

[14] R. J. R. Back and F. Kurki-Suonio. Distributed cooperation with action systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513–554, 1988.

[15] Bahareh Badban, Martin Fränzle, Jan Peleska, and Tino Teige. Test automation for hybrid systems. In *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*, pages 14–21, New York, NY, USA, 2006. ACM.

[16] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. *SIGSOFT Softw. Eng. Notes*, 14(8):210–218, 1989.

[17] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In Yolande Berbers and Willy Zwaenepoel, editors, *EuroSys*, pages 73–85. ACM, 2006.

[18] S. Barbey and D. Buchs. Testing Ada abstract data types using formal specifications. In *1st Int. Eurospace-Ada-Europe Symposium*, volume 887 of *Lecture Notes in Computer Science*, pages 76–89. Springer-Verlag, 1994.

[19] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[20] Mike Barnett, Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2005.

[21] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In *Poceedings of the International Symposium on Software Testing and Analysis*, 2008. to appear.

[22] Axel Belinfante, Lars Frantzen, and Christian Schallhart. Tools for test case generation. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 391–438. Springer, 2004.

[23] Eddy Bernard, Bruno Legeard, Xavier Luck, and Fabien Peureux. Generation of test sequences from formal specifications: Gsm 11-11 standard case study. *Softw., Pract. Exper.*, 34(10):915–948, 2004.

[24] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[25] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[26] Céline Bigot, Alain Faivre, Jean-Pierre Gallois, Arnault Lapitre, David Lugato, Jean-Yves Pierron, and Nicolas Rapin. Automatic test generation with agatha. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 591–596. Springer, 2003.

[27] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation Operators for Specifications. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, Washington, DC, USA, 2000. IEEE Computer Society.

[28] Mark R. Blackburn, Robert Busser, Aaron Nauman, Robert Knickerbocker, and Richard Kasuda. Mars polar lander fault identification using model-based testing. In *ICECCS*, pages 163–. IEEE Computer Society, 2002.

[29] MR Blackburn and RD Busser. T-VEC: a tool for developing critical systems. *Computer Assurance, 1996. COMPASS'96,'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*, pages 237–249, 1996.

[30] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In Jens Grabowski and Brian Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2004.

[31] J. Bohn, W. Damm, J. Klose, A. Moik, and H. Wittke. Modeling and validating train system applications using statemate and live sequence charts. In H. Ehrig, B. J. Kramer, and A. Ertas, editors, *Proceedings of the Conference on Integrated Design and Process Technology (IDPT2002)*, 2002.

[32] Henrik C. Bohnenkamp and Axel Belinfante. Timed testing with torx. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2005.

[33] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In Ramakrishnan and Rehof [186], pages 351–366.

[34] L. Bougé. *Modélisation de la notion de test de programmes, application à la production de jeux de test.* PhD thesis, Université de Paris 6, 1982.

[35] L. Bougé. A contribution to the theory of program testing. *Theoretical Computer Science*, 37(2):151–181, 1985.

[36] L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test set generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360, 1986.

[37] A. Bouwer, J. Liem, and B. Bredeweg. *User Manual for Single-User Version of QR Workbench*. Naturnet-Redime, STREP project co-funded by the European Commission within the Sixth Framework Programme (2002-2006), 2005. Project no. 004074. Project deliverable D4.2.1.

[38] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The if toolset. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 237–267. Springer, 2004.

[39] Harald Brandl, Gordon Fraser, and Franz Wotawa. Qr-model based testing. In *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, pages 20–17, New York, NY, USA, 2008. ACM.

[40] Harald Brandl and Franz Wotawa. Test case generation from qr models. In *21st International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems*, 2008.

[41] N. C. W. M. Braspenning, J. M. van de Mortel-Fronczak, and J. E. Rooda. A model-based integration and testing method to reduce system development effort. *Electr. Notes Theor. Comput. Sci.*, 164(4):13–28, 2006.

[42] N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, H.A.J. Neerhof, and J.E. Rooda. Model-based integration and testing in practice. In Jan Tretmans, editor, *Tangram: Model-based integration and testing of complex high-tech systems*. Embedded Systems Institute, Eindhoven, The Netherlands, 2007. chapter 7.

[43] B. Bredeweg, J. Liem, A. Bouwer, and P Salles. *Curriculum for learning about QR modelling*. Naturnet-Redime, STREP project co-funded by the European Commission within the Sixth Framework Programme (2002-2006), 2005. Project no. 004074. Project deliverable D6.9.1.

[44] Bert Bredeweg, Anders Bouwer, Jelmer Jellema, Dirk Bertels, Floris Floris Linnebank, and Jochem Liem. Garp3 - a new workbench for qualitative reasoning and modelling. In *Proceedings of 20th International Workshop on Qualitative Reasoning (QR-06)*, pages 21–28, Hannover, New Hampshire, USA, 2006.

[45] Manfred Broy, Franz Huber, and Bernhard Schätz. Autofocus - ein werkzeugprototyp zur entwicklung eingebetteter systeme. *Inform., Forsch. Entwickl.*, 14(3):121–134, 1999.

[46] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Verifying test-hypotheses – an experiment in test and proof. In Bernd Finkbeiner, Yuri Gurevich, and Alexander K. Petrenko, editors, *Model-based Testing (MBT 2008)*, Electronic Notes in Theoretical Computer Science, Budapest, Hungary, 2008. Elsevier Science Publishers. To appear.

[47] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[48] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *STTT*, 7(3):212–232, 2005.

[49] J. C., H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998.

[50] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.

[51] Jens R. Calame, Nicolae Goga, Natalia Ioustinova, and Jaco van de Pol. Ttcn-3 testing of hoornkersenboogerd railway interlocking. In *CCECE*, pages 620–623. IEEE, 2006.

[52] John Callahan, Francis Schneider, and Steve Easterbrook. Automated Software Testing Using Model-Checking. In *Proceedings 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.

[53] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis. Translating discrete-time simulink to lustre. In Rajeev Alur and Insup Lee, editors, *EMSOFT*, volume 2855 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2003.

[54] H.Y. Chen, T.H. Tse, and T.Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10(1):56–109, 2001.

[55] Kwang-Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *DAC*, pages 86–91, 1993.

[56] Yoonsik Cheon. Automated random testing to detect specification-code inconsistencies. In Dimitris A. Karras, Daming Wei, and Jaroslav Zendulka, editors, *SETP*, pages 112–119. ISRST, 2007.

[57] Yoonsik Cheon, Antonio Cortes, Martine Ceberio, and Gary T. Leavens. Integrating random testing with constraints for improved efficiency and diversity. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*. Department of Computer Science, The University of Texas at El Paso, February 2008. to appear.

[58] Yoonsik Cheon and Carlos E. Rubio-Medrano. Random test data generation for java classes annotated with jml specifications. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 385–391. CSREA Press, 2007.

[59] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A New Symbolic Model Verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.

[60] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. ACM Press, 2000.

[61] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 470–475. Springer, 2002.

[62] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[63] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Softw. Eng.*, 15(11):1318–1332, 1989.

[64] R. Cleaveland and A. E. Zwarico. A theory of testing for real-time. In *Proc. of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 110–119, Amsterdam, The Netherlands, 1991.

[65] Camille Constant, Bertrand Jeannet, and Thierry Jéron. Automatic test generation from interprocedural specifications. In Petrenko et al. [181], pages 41–57.

[66] EU FP5-GROWTH contract no G4RD-CT-2000-00361. Enhanced safety assessment for complex systems.

[67] Ian Craggs, Manolis Sardis, and Thierry Heuillard. Agedis case studies: Model-based testing in industry. In *Proceedings of the 1st European Conference on Model Driven Software Engineering*, pages 106–117, 2003.

[68] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In Robby, editor, *ICSE*, pages 281–290. ACM, 2008.

[69] J. C. Cunha, M. Z. Rela, and J. G. Silva. Can software implemented fault-injection be used on real-time systems? In *Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, pages 209–228, 1999.

[70] H. Dang Van, C. George, T. Janowski, and R. Moore, editors. *Specification case studies in RAISE*. Springer-Verlag, London, UK, 2002.

[71] Conrado Daws, Alfredo Olivero, and Sergio Yovine. Verifying et-lotos programmes with kronos. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 227–242, London, UK, UK, 1995. Chapman & Hall, Ltd.

[72] Luca de Alfaro. Game models for open systems. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 269–289. Springer, 2003.

[73] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[74] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In Ramakrishnan and Rehof [186], pages 337–340.

[75] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11:34–41, 1978.

[76] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[77] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[78] R.K. Dong and Ph.G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):103–130, 1994.

[79] Winfried Dulz and Fenhua Zhen. Matelo - statistical usage testing by annotated sequence diagrams, markov chains and ttcn-3. In *QSIC*, pages 336–342. IEEE Computer Society, 2003.

[80] Standard ecma–335, common language infrastructure (cli), June 2006.

[81] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 169–180, New York, NY, USA, 1982. ACM Press.

[82] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.

[83] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Ed Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, Enschede, the Netherlands, April 1997. Springer-Verlag.

[84] J.-C. Fabre, F. Salles, M. Rodríguez Moreno, and J. Arlat. Assessment of COTS microkernels by fault injection. In *Proceedings of the Conference on Dependable Computing for Critical Applications*, pages 25–44, 1999.

[85] Alain Faivre, Christophe Gaston, and Pascale Le Gall. Symbolic model based testing for component oriented systems. In Petrenko et al. [181], pages 90–106.

[86] P. Folkesson, S. Svensson, and J. Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, pages 284–293, 1998.

[87] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A symbolic framework for model-based testing. In *1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *LNCS*, pages 40–54. Springer, 2006.

[88] Gordon Fraser, Martin Weiglhofer, and Franz Wotawa. Coverage based testing with test purposes. In *Proceedings of the 8th International Conference on Quality Software*, 2008. to appear.

[89] Gordon Fraser, Martin Weiglhofer, and Franz Wotawa. Using observer automata to select test cases for test purposes. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, 2008. to appear.

[90] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *ISSTA*, pages 134–143, 2002.

[91] Peter Fritzson. Modelica - a language for equation-based physical modeling and high performance simulation. In Bo Kågström, Jack Dongarra, Erik Elmroth, and Jerzy Wasniewski, editors, *PARA*, volume 1541 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 1998.

[92] K. Futatsugi, J.A. Goguen, J.P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages, New Orleans*, pages 52–66, 1995.

[93] Stefan J. Galler, Robert Königshofer, Bernhard Peischl, Robert Unterberger, and Franz Wotawa. Automatic test generation tools for java based on design-by-contract(tm): A survey. Technical report, Competence Networt Softnet Austria, May 2008.

[94] Stefan J. Galler, Bernhard Peischl, and Franz Wotawa. Formal specification languages for design-by-contract in java: A survey. Technical report, Competence Network Softnet Austria, 2008.

[95] John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction implementation, specification and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.

[96] Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2007.

[97] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.

[98] Marie-Claude Gaudel. Testing can be formal too. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, May 1995.

[99] Marie-Claude Gaudel and Pascale Le Gall. Testing data types implementations from algebraic specifications. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing 2008*, volume 4949 of *LNCS*, pages 209–239, 2008.

[100] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[101] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. Technical Report MSR-TR-2007-58, Microsoft Research, May 2007.

[102] Nicolae Goga. Comparing torx, autolink, tgv and uio test algorithms. In Rick Reed and Jeanne Reed, editors, *SDL Forum*, volume 2078 of *Lecture Notes in Computer Science*, pages 379–402. Springer, 2001.

[103] K. K. Goswami, R. K. Iyer, and L. Young. DEPEND: A simulation-based environment for system level dependability analysis. *IEEE Transactions on Computers*, 46(1):60–74, 1997.

[104] Jens Grabowski and Dieter Hogrefe. The Standardization of Core INAP CS-2 by ETSI. Technical report, Technical Report A-99-02, Medical University of Lübeck, Schriftenreihe der Institute für Mathematik/Informatik, Lübeck, February 1999, February 1999.

[105] Wolfgang Grieskamp, Nicolas Kicillof, and Nikolai Tillmann. Action machines: a framework for encoding and composing partial behaviors. *International Journal of Software Engineering and Knowledge Engineering*, 16(5):705–726, 2006.

[106] Wolfgang Grieskamp and Carsten Weise, editors. *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*, volume 3997 of *Lecture Notes in Computer Science*. Springer, 2006.

[107] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice-Hall, 1992.

[108] The RAISE Method Group. *The RAISE Development Method*. The BCS Practitioners Series. Prentice-Hall, 1995.

[109] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

[110] J. Guthoff and V. Sieh. Combining software-implemented and simulation-based fault injection into a single fault injection method. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 196–206, 1995.

[111] J. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):369–405, 1977.

[112] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer Verlag, 1993.

[113] J.V. Guttag, J.J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, 1985.

[114] Daniel R. Hackner and Atif M. Memon. Test case generator for guitar. In Robby, editor, *ICSE Companion*, pages 959–960. ACM, 2008.

[115] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[116] S. Han, K.G. Shin, and H.A. Rosenberg. DOCTOR: An integrated software fault injection environment for distributed real-time systems. *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213, 1995.

[117] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[118] A. Hartman and K. Nagin. The agedis tools for model based testing. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 129–132. ACM, 2004.

[119] Alan Hartman. Final project report. Technical report, AGEDIS, February 2004.

[120] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. Uml-based integration testing. In *ISSTA*, pages 60–70, 2000.

[121] Jean Hartmann, Marlon Vieira, Herbert Foster, and Axel Ruder. A uml-based approach to system testing. *ISSE*, 1(1):12–24, 2005.

[122] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[123] T. A. Henzinger. The theory of hybrid automata. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 278, Washington, DC, USA, 1996. IEEE Computer Society.

[124] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In Hierons et al. [128], pages 77–117.

[125] Anders Hessel and Paul Pettersson. A test case generation algorithm for real-time systems. In *QSIC '04: Proceedings of the Quality Software, Fourth International Conference*, pages 268–273, Washington, DC, USA, 2004. IEEE Computer Society.

[126] Anders Hessel and Paul Pettersson. Model-based testing of a wap gateway: An industrial case-study. In Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, editors, *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2006.

[127] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 2008.

[128] Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors. *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*. Springer, 2008.

[129] M. Hiller. *A Software Profiling Methodology for Design and Assessment of Dependable Software*. PhD thesis, Chalmers University of Technology, 2002.

[130] CAR Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[131] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *Lecture Notes in Computer Science*, pages 209–213, Princeton, NJ, USA, July 2008. Springer.

[132] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[133] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.

[134] Ivo Van Horebeek and Johan Lewi. *Algebraic Specifications in Software Engineering, an Introduction*. Springer Verlag, 1993.

[135] IEEE. Ieee standard for information technology - requirements and guidelines for test methods specifications and test method implementations for measuring conformance to posix(r) standards. *IEEE Std 2003-1997*, pages –, 2 Sep 1998.

[136] IEEE-ISTO. The nexus 5001(tm) forum standard for a global embedded processor debug interface, 1999. pp.9-10.

[137] J. Isacson and M. Ljungberg. Fault injection in Matlab/Simulink. Master's thesis, Chalmers University of Technology, 2008.

[138] ISO. ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989.

[139] ISO. Information processing systems — Open systems interconnection — Estelle — a formal description technique based on an extended state transition model, 1997.

[140] ISO/IEC. *Z formal specification notation – Syntax, type system and semantics*. ISO/IEC, 2002.

[141] ITU-T. Formal description techniques (fdt) – specification and description language (sdl), 2002.

[142] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[143] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, August 2005.

[144] Bertrand Jeannet, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Symbolic test selection based on approximate analysis. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2005.

[145] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *Proceedings of the 24th International Symposium on Fault Tolerant Computing*, pages 66–75, 1994.

[146] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[147] A. Joshi and M. P. E. Heimdahl. Model-based safety analysis of Simulink models using SCADE design verifier. In *Proceedings of the 24th International Conference on Computer Safety, Reliability and Security*, pages 122–135, 2005.

[148] A. Agung Julius, Georgios E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust test generation and coverage for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2007.

[149] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, 1995.

[150] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. Application of three physical fault injection techniques to the experimental assessment of the mars architecture. In *Proceedings of 5th IFIP Working Conference on Dependable Computing for Critical Applications*, pages 267–287, 1995.

[151] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1):8–23, 1994.

[152] Nicolas Kicillof, Wolfgang Grieskamp, Nikolai Tillmann, and Victor Braberman. Achieving both model and code coverage with automated gray-box testing. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 1–11, New York, NY, USA, 2007. ACM.

[153] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie-Mellon University, 1992.

[154] Johan De Kleer and John Seely Brown. A qualitative physics based on confluences. *Artif. Intell.*, 24(1-3):7–83, 1984.

[155] Beat Koch, Jens Grabowski, Dieter Hogrefe, and Michael Schmitt II. Autolink: A tool for automatic test generation from sdl specifications. In *WIFT*, pages 114–. IEEE Computer Society, 1998.

[156] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[157] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal: Status and future work. In Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans, editors, *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http://drops.dagstuhl.de/opus/volltexte/2005/326> [date of citation: 2005-01-01].

[158] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306, New York, NY, USA, 2005. ACM.

[159] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[160] David Lugato, Frédéric Maraux, Yves Le Traon, Véronique Normand, Hubert Dubois, Jean-Yves Pierron, Jean-Pierre Gallois, and Clémentine Nebut. Automated functional test case synthesis from thales industrial requirements. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 104–111. IEEE Computer Society, 2004.

[161] H. Madeira, M. Z. Rela, F. Moreira, and J. G. Silva. RIFLE: A general purpose pin-level fault injector. In *Proceedings of the 1st European Dependable Computing Conference*, pages 199–216, 1994.

[162] K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. PhD thesis, Eindhoven University of Technology, 2006.

[163] E. Martins and A. C. A. Rosa. A fault injection approach based on reflective programming. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 407–416, 2000.

[164] E. Martins, C. M. F. Rubira, and N. G. M. Leme. Jaca: A reflective fault injection tool based on patterns. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 483–487, 2002.

[165] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[166] Atif M. Memon. An event-flow model of gui-based applications for testing. *Softw. Test., Verif. Reliab.*, 17(3):137–157, 2007.

[167] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000.

[168] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1982.

[169] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[170] Andreas Moser, Christopher Krügel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245. IEEE Computer Society, 2007.

[171] Peter D. Mosses. CASL: A guided tour of its design. In *Proceedings of WADT'98*, volume 1589 of *Lecture Notes in Computer Science*, pages 216–240. Springer-Verlag, 1999.

[172] Giuseppe De Nicola, Pasquale di Tommaso, Rosaria Esposito, Francesco Flammini, Pietro Marmo, and Antonio Orazzo. A grey-box approach to the functional testing of complex automatic train protection systems. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *EDCC*, volume 3463 of *Lecture Notes in Computer Science*, pages 305–317. Springer, 2005.

[173] Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In *Proceedings of the LITP spring school on theoretical computer science on Semantics of systems of concurrent processes*, pages 407–419, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[174] Manuel Núñez and Ismael Rodríguez. Conformance testing relations for timed systems. In Grieskamp and Weise [106], pages 103–117.

[175] Test Technology Technical Committee of the IEEE Computer Society. IEEE standard test access port and boundary-scan architecture. Technical report, 1990.

[176] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for circus. *Electr. Notes Theor. Comput. Sci.*, 187:107–123, 2007.

[177] The Object Management Group (OMG). Omg systems modeling language.

[178] Norbert Oster. *Automatische Generierung optimaler struktureller Testdaten fÃijr objekt-orientierte Software mittels multi-objektiver Metaheuristiken*. PhD thesis, Friedrich–Alexander University, Erlangen-NÃijrnberg, 2007.

[179] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia. A visual test development environment for gui systems. *SIGSOFT Softw. Eng. Notes*, 23(2):82–92, 1998.

[180] Paul. Testing with model checkers: A survey. Technical Report SNA-TR-2007-P2-04, Competence Network Softnet Austria, 2007.

[181] Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors. *Testing of Software and Communicating Systems, 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007, Proceedings*, volume 4581 of *Lecture Notes in Computer Science*. Springer, 2007.

[182] Alexandre Petrenko and Nina Yevtushenko. Queued testing of transition systems with inputs and outputs. In Rob Hierons and Thierry Jéron, editors, *Proceedings of the Workshop on Formal Approaches to Testing Of Software (Fates'02), A Satellite Workshop of Concur'02*, pages 79–94, Brno, Czech Republic, August 2002.

[183] Alexandre Petrenko and Nina Yevtushenko. Conformance tests as checking experiments for partial nondeterministic fsm. In Grieskamp and Weise [106], pages 118–133.

[184] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA*, pages 46–57. IEEE, 1977.

[185] EU FP6-AEROSPACE project reference 501848. Improvement of safety activities on aeronautical complex systems.

[186] C. R. Ramakrishnan and Jakob Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[187] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, 11(4):367–375, 1985.

[188] Inc. Reactive Systems. Model-based testing and validation with reactis. Technical report, Reactive Systems, Inc., 2003.

[189] M. Rebaudengo and M. Reorda. Evaluating the fault tolerance capabilities of embedded systems via BDM. In *Proceedings of the 17th IEEE VLSI Test Symposium*, pages 452–457, 1999.

[190] Mark Richters and Martin Gogolla. On formalizing the uml object constraint language ocl. *Conceptual Modeling –ER '98*, pages 449–464, 1998.

[191] Tomas Rokicki and Chris J. Myers. Automatic verification of timed circuits. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 468–480, London, UK, 1994. Springer-Verlag.

[192] Mauno Rönkkö, Anders P. Ravn, and Kaisa Sere. Hybrid action systems. *Theor. Comput. Sci.*, 290(1):937–973, 2003.

[193] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison–Wesley, 2nd edition, 2004.

[194] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002.

[195] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *IFM*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer, 2000.

[196] Ina Schieferdecker, Jens Grabowski, Theofanis Vassiliou-Gioles, and George Din. The test technology ttcn-3. In Hierons et al. [128], pages 292–319.

[197] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[198] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.

[199] V. Sieh, O. Tschäche, and F. Balbach. VERIFY: Evaluation of reliability using vhdl-models with embedded fault descriptions. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 32–36, 1997.

[200] S. Sims and D.C. DuVarney. Experience report: the reactis validation tool. *Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 137–140, 2007.

[201] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. Evaluating the cost of software quality. *Commun. ACM*, 41(8):67–73, 1998.

[202] Li Tan, Oleg Sokolsky, and Insup Lee. Specification-based testing with linear temporal logic. In *Proceedings of IEEE International Conference on Information Reuse and Integration (IRI'04)*, pages 493–498, 2004.

[203] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[204] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 127–146, London, UK, 1996. Springer-Verlag.

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

[206] 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.

[207] Jan Tretmans and Ed Brinksma. TorX: Automated model based testing. In A. Hartman and K. Dussa-Zieger, editors, *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 13–25, Nurnburg, Germany, 2003.

[208] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[209] Michiel van Osch. Hybrid input-output conformance and test generation. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2006.

[210] Daniel D. Vanzin, Ivan L. Martins, and Joao Bosco A. Pereira Filho. Tde uml editor - a success development case of a software extension. In *ICGSE '06: Proceedings of the IEEE international conference on Global Software Engineering*, pages 257–258, Washington, DC, USA, 2006. IEEE Computer Society.

[211] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Hierons et al. [128], pages 39–76.

[212] J. Vinter, J. Aidemark, D. Skarin, R. Barbosa, P. Folkesson, and J. Karlsson. An overview of GOOFI - a generic object-oriented fault injection framework. Technical Report 05-07, Department of Computer Science and Engineering, Chalmers University of Technology, 2005.

[213] J. Vinter, L. Bromander, P. Raistrick, and H. Edler. FISCADE - a fault injection tool for SCADE models. In *Proceedings of the 3rd IET Conference on Automotive Electronics*, pages 1–9, 2007.

[214] S. Vulinovic and B.H. Schlingloff. Model based dependability evaluation for automotive control functions. In *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics*, 2005.

[215] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[216] Martin Weiglhofer and Franz Wotawa. "On the fly" input output conformance verification. In *Proceedings of the IASTED International Conference on Software Engineering*, Innsbruck, Austria, February 2008. To appear.

[217] Franz Wotawa. Generating test-cases from qualitative knowledge – preliminary report. In *Proceedings of the 21st Annual Workshop on Qualitative Reasoning*, Aberystwyth, U.K., June 2007.

[218] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1), 2007.

[219] P. Yuste, D. de Andres, L. Lemus, J.J. Serrano, and P. Gil. INERTE: Integrated nexus-based real-time fault injection tool for embedded systems. *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, page 669, 2003.