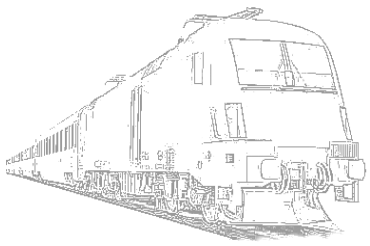
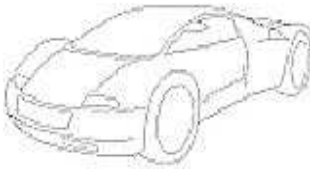
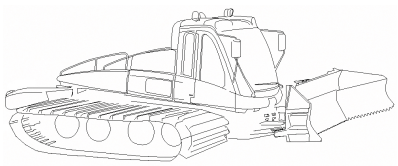
MOGENTES

Model-Based Generation of Test-Cases for Embedded Systems

Fault-based Test-Case Generation Methods

Project	MOGENTES		**Contract Number**		216679
Document Id	4-01_1.0r	**Date**	2008-12-30	**Deliverable**	D 4.1
Contact Person	Georg Weissenbacher			**Organisation**	ETH Zurich
Phone	+44 1865 283518			**E-Mail**	georg.weissenbacher @inf.ethz.ch

Distribution Table

Name	Company	Department	No. of copies	Hard/Softcopy
all MOGENTES team members				

Change History

Version	Date	Reason for Change	PagesAffected
0.1w	2008-12-08	first version (internal)	all
0.2w	2008-12-23	integrated comments from ARCS and SP	all
1.0r	2008-12-30	integrated comments from ARCS and SP	all

Contents

1 Introduction **4**
1.1 Purpose and Scope . . . 5

2 Model Checking **6**
2.1 Bounded Model Checking . . . 6
2.1.1 Background on BMC . . . 6
2.1.2 Unwinding the Entire Program at Once . . . 7
2.1.3 Unwinding Loops Separately . . . 8
2.1.4 A Complete BMC for Software . . . 9
2.1.5 BMC and k-Induction. . . . 9
2.1.6 Solving the Decision Problem . . . 9
2.1.7 Merits and Shortcomings of BMC . . . 10
2.2 Equivalence Checking . . . 10

3 Test-Case Generation Using Model Checking **12**
3.1 Outline of the Problem . . . 12
3.1.1 Availability of a Model . . . 12
3.1.2 Quality of the Test Suite . . . 12
3.2 Generating Test Cases with Model Checkers . . . 13

4 Fault-Based Test Case Generation **15**
4.1 Mutation Testing and Fault Injection . . . 15
4.2 Generating Test Cases . . . 16
4.2.1 Encoding Combinations of Faults . . . 16
4.2.2 Searching for Combinations of Faults . . . 17
4.2.3 Generating Test Cases for Continuous Models . . . 18
4.2.4 Generating Test Cases for Stress Testing . . . 20
4.3 Issues Specific to the Modelling Language . . . 20

5 Abbreviations and Definitions **22**

1 Introduction

Despite the advances of formal verification techniques, testing is still the prevailing approach in system verification. One reason is that verification engineers have many years of experience with testing, and that this technique has earned their trust. Test cases are simple to understand, while formal proofs or specifications are complex constructs and therefore often less convincing. Another reason are the scalability issues automated verification techniques still have not overcome, while testing, once the test suite has been created, imposes almost no limitations on the size of the system.

Creating a useful test suite, however, can be a cumbersome and time consuming task. The MOGENTES project aims at using state-of-the-art automated verification techniques (such as Model Checking) to extract test suites from the system model. Furthermore, the objective is to create an *efficient* test suite, keeping the number of test cases necessary to reach the desired coverage low. The approach we propose has three main aspects:

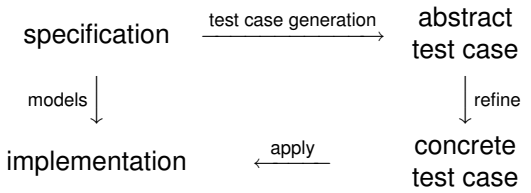
a) It is *model driven.* Model Driven Development (MDD) hides the platform-specific details of the implementation by using modelling languages that are more abstract than traditional implementation languages such as C or C++. The model comprises components that are eventually implemented in either hardware or software in the actual system.

b) It is *fault based.* While the adequacy of a test suite is traditionally defined by means of coverage metrics such as statement coverage or modified condition/decision coverage, we evaluate our test suite by checking whether it covers a predefined set of conceivable faults. We achieve this by systematically injecting faults into the model of the system.[1] The motivation for choosing this approach is that we want to find test cases that detect a certain class of faults in the system.

c) It is based on *automated verification techniques.* In order to reduce the time spent on system verification, we apply automated verification techniques to generate test cases. While Model Checking may not be efficient enough to verify entire implementations, it has been shown be scalable enough to generate test cases for medium-sized implementations [Ball, 2005; Beyer *et al.*, 2004; Holzer *et al.*, 2009] and for models of embedded systems [Micskei and Majzik, 2006].

In this report, we distinguish the following two scenarios:

1. The model serves as a (partial) specification for the implementation, reflecting the requirements of the product, and thus as an *oracle* for the test case generation. In this scenario, the implementation, or at least parts of the implementation, are *not* automatically generated from the model. In that case, it is necessary to verify the consistency of the model and the implementation. A formal proof of consistency is often infeasible. Therefore, we aim at extracting a representative set of test cases from the model that allows us to perform a "point-wise" verification of the implementation. If the set of test vectors is "sufficiently dense", it can serve as a convincing argument that the implementation adheres to its specification (i.e., the model). The following commutative diagram depicts the situation described above:

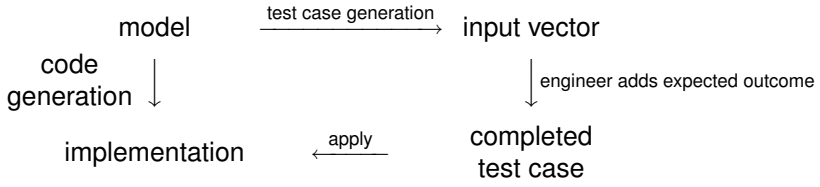


Since the "models" relation between the specification and the implementation cannot be efficiently verified, we revert to traditional testing. Using Model Checking, we single out execution traces of the model that satisfy the given coverage metrics. Efficient decision procedures enable the extraction of test vectors from these execution traces to obtain a test suite. The model acts as an *oracle* for the test cases, i.e., it defines the outcome for each test case. Since the resulting test vectors are described in the context of the model, it is necessary to refine them so that they can be applied to the implementation.

2. The model *is* the implementation and the actual executable code is automatically generated from the model (or from the source code of the implementation). In that case, the model cannot be used as an oracle for the generated test cases, since the implementation and the model are consistent by construction

[1] Since the model may also include a description of the environment, it is possible to represent *external* faults such as incorrect usage of the system, as well as *internal* faults such as malfunctioning components.

(assuming the correctness of the code generator). We are still able to extract tests that satisfy the given coverage metrics, but deriving the outcome of the test cases from the respective executions would yield a test suite that trivially succeeds. In that case, the test engineer is required to act as an oracle, i.e., he has to specify appropriate test results or to observe the test run and check whether the results comply with the expected behaviour. This situation is depicted in the following commutative diagram:



The coverage metrics we focus on in this deliverable is based on code mutations and (model/software/hardware implemented) fault injection. The test cases that are generated based on the mutations of the source code provide a structural coverage of the software, while the test cases generated by injecting hardware faults (either in the model, the software, or physically) can measure the reliability of the system under test. This decision is motivated by the fact that we want to verify dependable embedded systems. In this domain, the reliability of a system is commonly tested by injecting faults into the implementation and checking how the system behaves in the presence of these faults. A typical requirement is that the system should be able to tolerate at least a single fault. Our aim is to generate test cases that reveal the erroneous behaviour in case that the implementation fails to cope with a fault. A more detailed description of the fault models we consider is provided in the MOGENTES Deliverable D3.1a.

1.1 Purpose and Scope

This document provides a generic description of the algorithms and techniques we have devised to extract fault based test cases from models. Since the detailed specification of the modelling language (MOGENTES deliverables D3.2a and D3.2b) and the final list of fault models (Deliverable D3.1b) are only due in month 18 and month 24, respectively, we decided to keep the presentation of the algorithms generic and independent from the modelling language. This document does not provide a detailed discussion of fault models, since this topic is covered by Deliverable D3.1a.

This document is structured as follows: We introduce the automated verification techniques underlying our test case generation approach in Chapter 2. In Chapter 3, we explain how these techniques may be used to generate test cases, starting with an overview of the issues that arise in automated test case generation (Chapter 3.1), and concluding with an overview over existing approaches (Chapter 3.2). Chapter 4 contains a presentation of the algorithms that we use to generate test cases based on mutations and fault injection. We discuss how we integrate mutations and faults into the model in Chapter 4.1. Based on the resulting modified model, we present our test case generation approach in Chapter 4.2. This section also contains a discussion of the extensions to our decision procedure that are necessary do deal with values of a continuous domain (i.e., “analogue” values). In Chapter 4.2.4, we explain how test cases that put the system under stress can be generated. Finally, we discuss modelling language-specific issues in Chapter 4.3, based on the example of the Simulink language.

2 Model Checking

Model Checking, in the most general sense, is a technique that explores the reachable states of a model in order to determine whether a given specification is satisfied [Clarke *et al.*, 1999]. It differs from testing in so far as it aims at covering the entire state space of the model or program under test, providing a correctness guarantee that is rarely achieved by means of testing. If the specification is violated, Model Checking tools are often able to provide a *counterexample*, i.e., a witness for the incorrectness of the model. This ability to report counterexamples is the essential feature we use to generate test cases (see Chapter 3). The disadvantage of Model Checking is that it does not scale as well as testing.

In our setting, we deal exclusively with specifications stating simple properties such as the reachability of statements. As we explain in Chapter 3, this is sufficient to enable the generation of test suites that satisfy the most common coverage criteria.

A detailed description of the modelling language underlying the MOGENTES project will be provided in the deliverable D3.2a, due in M 18. For this report, we assume the existence of a behavioural model, specified by a (possibly partial) transition relation R and a predicate I that determines the valid initial states of the model. The transition relation R relates the current state of the model to its successor states (i.e., the potential states after one step). The structure of R may be further detailed by means of a *control flow graph* (see, for instance, Figure 1). The control flow graph partitions R into local transition functions, each of which corresponds to one edge of the graph. Each state of the model comprises a location (i.e., it is associated to a node of the control flow graph) and a valuation to the variables or signals of the model. A local transition function can only be activated if its corresponding edge is an outgoing edge of the current location. Note that this model is sufficiently general to allow imperative models (such as C programs or state charts) as well as data flow models (such as Simulink models). Furthermore, the predicate I characterises the set of valid initial states of the model (this may be the safe or reset state of the system), i.e., $I(s)$ holds if s is a valid initial state.

A *path* (or execution trace) π of the model is a sequence of states $s_0, s_1, \ldots, s_n$ such that the adjacent pairs s_i, s_{i+1} of states in that sequence are related by R (i.e., $\bigwedge_{i=0}^{n-1} R(s_i, s_{i+1})$), and $I(s_0)$ holds (i.e., s_0 is a valid initial state). A state is induced by the values of the variables (or wires) of the model. In reactive models, the variables are typically partitioned into input variables, hidden (or internal) variables, and output variables. The observable part of an execution trace is therefore the sequence of inputs and the resulting sequence of outputs. Given a state s_i, we use $s_i.i$ to refer to the input, and $s_i.o$ to denote the output.

In its simplest incarnation, a Model Checker is a tool that exhaustively examines the reachable states in order to determine whether *bad* states are reachable. This is achieved by simulating the feasible execution traces of the model (for all possible inputs). The execution trace currently examined can be pruned if it reaches a state that has been already visited. For this purpose, the Model Checking tool has to keep track of the visited states. *Explicit state Model Checking* tools use an explicit representation of the states reached during the search and are conceptually close to testing. *Symbolic Model Checking* tools represent states by means of formulas, a representation that can be exponentially more succinct than an explicit enumeration of states [McMillan, 1993]. The set of valid assignments to a formula $F(s_0)$ corresponds to a set of states, i.e., $F(s_0)$ characterises the set of states s_0. For instance, the formula

$$F(s_2) := \exists s_0, s_1 \,.\, I(s_0) \wedge R(s_0, s_1) \wedge R(s_1, s_2)$$

characterises the set of states reachable after two execution steps. The implicit representation of states makes it harder to determine whether a given set of states has already been explored, since a decision procedure is required to compute the models[2] of (i.e., the states described by) the formula F.

The *fixed-point* detection, i.e., the process of determining whether the states currently examined have already been explored, is often the bottle-neck of Model Checking. In the following section, we describe a technique that avoids fixed-point detection at the cost of completeness (i.e., the approach may "miss" states).

2.1 Bounded Model Checking

2.1.1 Background on BMC

Bounded model checking (BMC) is one of the most commonly applied formal verification techniques in the semiconductor industry. The technique owes this success to the impressive capacity of propositional SAT solvers. It was introduced in 1999 by Biere et al. as a complementary technique to BDD-based unbounded model checking [Biere *et al.*, 1999]. It is called *bounded* because only states reachable within a bounded

[2] In this context, the term *model* refers to a model of a first order logic formula, i.e., a valid assignment to all variables of the formula.

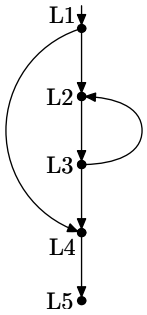


Figure 1: A Control Flow Graph

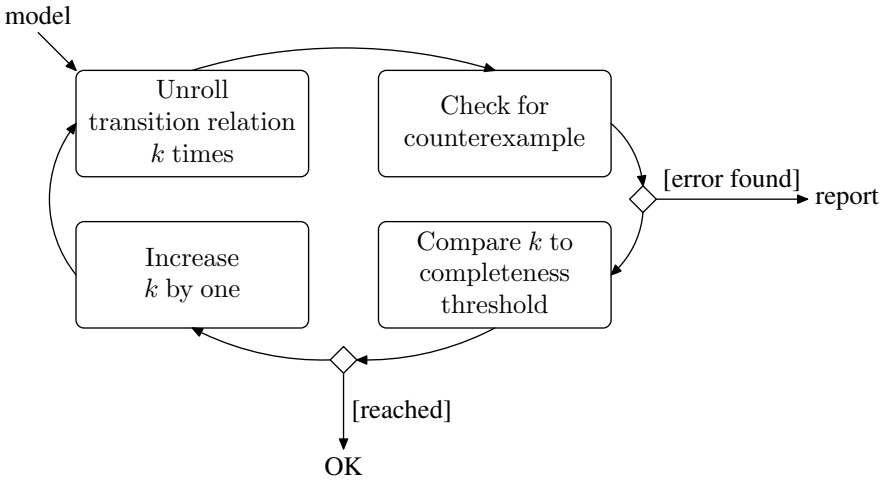
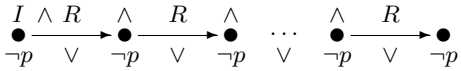


Figure 2: High level overview of BMC

number of steps, say k, are explored. In BMC, the design under verification is unwound k times and conjoined with a property to form a propositional formula, which is passed to a SAT solver (Fig. 2). The formula is satisfiable if and only if there is a trace of length k that refutes the property. Thus, a satisfying assignment to the formula corresponds to a *counterexample*. The technique is inconclusive if the formula is unsatisfiable, as there may be counterexamples longer than k steps. Nevertheless, the technique is successful, as many bugs have been identified that would otherwise have gone unnoticed.

Let R denote the transition relation of a design, containing current and next state pairs, I denote the initial state predicate, and p denote the property of interest. To obtain a BMC instance with k steps, the transition relation is replicated k times. The variables in each replica are renamed such that the next state of step i is used as current state of step $i + 1$. The transition relations are conjoined, the current state of the first step is constrained by I, and one of the states must satisfy $\neg p$:

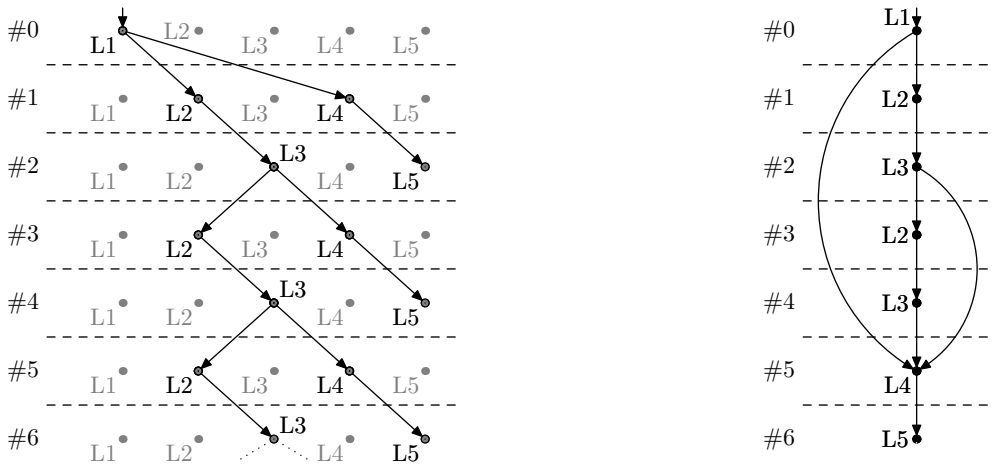


A satisfying assignment for this formula corresponds to a path from the initial state to a state violating p. The size of this formula is linear in the size of the design and in k.

We describe BMC for software in Chapter 2.1.2 and optimisations to deal with loops in Chapter 2.1.3. BMC is usually not *complete*, that is, it cannot be used to prove and disprove properties, as we discuss in Chapter 2.1.4. In Chapter 2.1.5, we discuss k-induction, which is a variation of BMC that addresses this problem in some cases. In Chapter 2.1.6, we survey decision procedures for formulas resulting from BMC, and conclude this section with a discussion in Chapter 2.1.7.

2.1.2 Unwinding the Entire Program at Once

BMC is also applicable to system-level software. The most straight-forward manner to implement BMC for software is to treat the entire program as a transition relation as described in Chapter 2. Each basic block



(a) Unrolling the Transition Relation (b) Unrolling the Loops

Figure 3: There are two ways to unroll the model given by the control flow graph: When unrolling the entire transition relation (a), the unreachable nodes (in gray) may be omitted. When unrolling the loops (b), a separate unwinding bound has to be provided for each loop.

```
while(x)                    if(x) {
  BODY;          ⟶           BODY;
                              if(x)
                                BODY;
                              else
                                assume(false);
                            }
```

Figure 4: A loop-based unrolling of a **while** loop with depth two. The assume statement cuts of any path passing through it.

is converted into a formula by transforming it into Static Single Assignment (SSA) form [Alpern *et al.*, 1988]. Arrays and pointers can be supported by a suitable choice for the logic the formulas are built with.

An unwinding with k steps permits exploring all program paths of length k or less. The size of this basic unwinding is k times the size of the program. For large programs, this is prohibitive, and thus, several optimisations have been proposed. The first step is to analyse the possible control flow in the program. Consider the small control flow graph in Fig. 1. Each node corresponds to one basic block; the edges correspond to possible control flow between the blocks. Note that block L1 can only be executed in the very first step of any path. Similarly, block L2 can only be executed in step 2, 4, 6 and so on. This is illustrated in Fig. 3(a): the unreachable nodes in each timeframe are in gray.

2.1.3 Unwinding Loops Separately

Consider again the example in Fig. 1: observe that any path through the CFG visits L4 and L5 at most once. Note that the unwinding of the transition relation in Fig. 3(a) contains three copies of L4 and L5. Such redundancy can be eliminated by building a formula that follows a specific path of execution rather than treating the program as a transition relation. In 2000, Currie et al. proposed this approach in a tool that unwinds assembly programs running on DSPs [Currie *et al.*, 2000]

This motivated the idea of *loop unwinding*. Instead of unwinding the entire transition relation, each loop is unwound separately. Syntactically, this corresponds to a replication of the loop body together with an appropriate guard (Fig 4). The effect on the control flow graph is illustrated in Fig. 3(b), in which the loop between L2 and L3 is unwound twice. Such an unwinding may result in more compact formulas and requires fewer case splits in the formula, as there are fewer successors for each timeframe. As a disadvantage, the loop-based unwinding may require more timeframes to reach certain locations. In Fig 3(a), the unwinding of depth 1 suffices to determine if $L4$ is reachable in one step, while an unwinding of depth 5 is required in Fig 3(b).

Loop unwinding differs from enumerative, path-based exploration: in the example, the path that corresponds to the branch from L1 to L4 merges with the path that follows the loop. As a consequence, the formula that is

generated is linear in the depth and in the size of the program, even though there is an exponential number of paths through the CFG.

2.1.4 A Complete BMC for Software

Bounded Model Checking, when applied as described above, is inherently incomplete, as it searches for property violations only up to a given bound and never returns "`No Errors`". Bugs that are deeper than the given bound are missed. Nevertheless, BMC can be used to *prove* liveness and safety properties if applied in a slightly different way.

Intuitively, if we could search *deep enough*, we could guarantee that we have examined all the relevant behaviour of the model, and that searching any deeper only exhibits states that we have explored already. A depth that provides such a guarantee is called a *completeness threshold* [Kroening and Strichman, 2003]. Computing the smallest such threshold is as hard as model checking, and thus, one settles for over-approximations in practice.

In the context of software, one way to obtain a depth-bound for a program is to determine a high-level worst-case execution time (WCET). This time is given by a bound on the maximum number of loop iterations and is usually computed via a simple syntactic analysis of loop structures. If the syntactic analysis fails, an iterative algorithm can be applied. First, a guess of the bound on the number of loop iterations is made. The loop is then unrolled up to this bound, as in Fig 4, but with the assumption replaced by an assertion called an *unwinding assertion*. If the assertion is violated, there are paths in the program exceeding the bound, and a new guess for the bound is made [Kroening *et al.*, 2003; Clarke *et al.*, 2004].

This method is applicable if the program (or its main loop body) has a runtime bound, which is highly desirable for many embedded applications.

2.1.5 BMC and k-Induction.

Another technique to obtain safety guarantees using BMC is k-induction. This technique aims at establishing an *inductive invariant* [Sheeran *et al.*, 2000]. The approach consists of three steps:

1. Show that the property holds in the first k steps, starting from the initial state:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \implies \bigwedge_{i=0}^{k} p(s_i) \tag{1}$$

2. Furthermore, let $\mathrm{loopFree}(s_0, \ldots, s_n)$ denote a path that does not visit a state repeatedly, i.e.,

$$\mathrm{loopFree}(s_0, \ldots, s_n) := \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1}) \wedge \bigwedge_{0 \le i < j \le n} s_i \neq s_j \,. \tag{2}$$

   Now, if it holds that any loop-free execution of length k starting in a state in which p holds does not violate p and guarantees that the property will also hold in the $(k+1)^{st}$ step, then the p holds in all reachable states by induction. This can be checked by testing whether the implication

$$\mathrm{loopFree}(s_0, \ldots, s_{k+1}) \wedge \bigwedge_{i=0}^{k-1} p(s_i) \implies p(s_{k+1}) \tag{3}$$

   holds.

The k-induction technique is particularly useful to show that an error does not propagate to the output. For instance, consider a bit-flip fault induced in the most significant bit of a 32-bit integer variable a. This fault has no impact on a condition that tests whether a is even or odd. Properties of this kind can be easily shown using k-induction.

In the next section, we discuss methods for deciding the formulas arising in BMC for software.

2.1.6 Solving the Decision Problem

The result of unwinding, either of the entire program, by unwinding loops, or by following specific paths of execution, is a bit-vector formula. In addition to the usual arithmetic and bit-level operators, the formula may

also contain operators related to pointers and (possibly unbounded) arrays. There is a large body of work on efficient solvers for such formulas. The earliest work on deciding bit-vector arithmetic is based on algorithms from the theorem proving community, and uses a canonizer and solver for the theory. The work by Cyrluk et al. [Cyrluk *et al.*, 1997] and by Barrett et al. on the Stanford Validity Checker [Barrett *et al.*, 1998] fall into this category. These approaches are very elegant, but are restricted to a subset of bit-vector arithmetic comprising concatenation, extraction, and linear equations (not inequalities) over bit-vectors.

With the advent of efficient propositional SAT solvers such as ZChaff [Moskewicz *et al.*, 2001], these approaches have been obsoleted. The most commonly applied approach to check satisfiability of these formulas is to replace the arithmetic operators by circuit equivalents to obtain a propositional formula, which is then passed to a propositional SAT solver. This approach is commonly called 'bit-blasting' or 'bit-flattening', as the word-level structure is lost. The Cogent [Cook *et al.*, 2005] procedure belongs to this category. The current version of CVC-Lite [Berezin *et al.*, 2005] pre-processes the input formula using a normalisation step followed by equality rewrites before finally bit-blasting to SAT. Wedler et al. [Wedler *et al.*, 2005] have a similar approach wherein they normalise bit-vector formulas to simplify the generated SAT instance. STP [Cadar *et al.*, 2006], which is the engine behind EXE [Yang *et al.*, 2006], is a successor to the CVC-Lite system; it performs several array optimisations, as well as arithmetic and Boolean simplifications on a bit-vector formula before bit-blasting. Yices [Dutertre and de Moura, 2006] applies bit-blasting to all bit-vector operators except for equality. Bryant et al. present a method to solve hard instances that is based on iterative abstraction refinement [Bryant *et al.*, 2007]. This technique has been implemented in commercial software as well, e.g., Synopsys' Hector.

In order to be able to deal with analogue signals, we need to provide a decision procedure for a combination of bit-vector and *floating point arithmetic*, which is predominantly used to process analogue values. Our prototype of a decision procedure for floating point arithmetic is presented in Chapter 4.2.3.

2.1.7 Merits and Shortcomings of BMC

BMC is the best technique to find shallow bugs, and it provides a full counterexample trace in case a bug is found. It supports the widest range of program constructions. This includes dynamically allocated data structures; for this, BMC does not require built-in knowledge about the data structures the program maintains. On the other hand, completeness is only obtainable on very 'shallow' programs, i.e., programs without deep loops, or specific properties that are amenable to k-induction.

BMC can be used to single out finite-length execution traces that have a given set of properties. In Chapter 3, we discuss how coverage criteria such as decision coverage or MC/DC can be encoded as reachability properties and how BMC can be used to derive a test-suite achieving this coverage.

Our ultimate goal, however, is to define the coverage of test-suites in terms of the number of *injected faults* it detects. For this purpose, we need to be able to find paths that uncover semantic modifications to the original model. Starting from a model M (defined by means of its transition function R), we introduce faults and mutations to obtain a modified model M' (a transition function R'). A "good" test vector is able to distinguish M and M', i.e., it yields different outcomes for the original and the modified/mutated model. We discuss a technique that allows us to obtain such test vectors in Chapter 2.2.

2.2 Equivalence Checking

(Formal) Equivalence Checking is a technique used to formally prove that two models M and M' exhibit the same observable behaviour [Kuehlmann and van Eijk, 2002]. This is achieved by comparing the input and the output behaviour of the two models. We classify equivalence checking techniques according to taxonomy presented by Godlin and Strichman [Godlin, 2008]:

1. **Partial Equivalence.** Given the same inputs, any two terminating executions of the models yield the same outputs.
2. **Mutual Termination.** Given the same input, either the execution of both models is terminating, or both executions don't terminate.
3. **Reactive Equivalence.** Given the same input sequences, both models emit equivalent output sequences. Reactive equivalence generalises partial and mutual equivalence.
4. **k-Equivalence.** Given the same input sequence, the outputs in the first k steps of the executions of the models match. This notion of equivalence is decidable, assuming that the input and output values have a finite range.

5. **Total Equivalence.** The programs are partially equivalent and always terminate.
6. **Full Equivalence.** The programs are partially equivalent and mutually terminate.

Whether two given models are k-equivalent can be decided using BMC. Given two models M and M' (comprising the transition functions R and R' and the initial state predicates I and I', respectively), we can check (assuming states of finite size) whether

$$\underbrace{\bigwedge_{i=0}^{k} s_i.i = s'_i.i}_{\text{equality of all inputs}} \wedge \underbrace{I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})}_{\text{first model}} \wedge \underbrace{I'(s'_0) \wedge \bigwedge_{i=0}^{k-1} R'(s'_i, s'_{i+1})}_{\text{second model}} \wedge \underbrace{\bigvee_{i=0}^{k} s_i.o \neq s'_i.o}_{\text{inequality of an output}} \tag{4}$$

is satisfiable. Any satisfying assignment to this formula represents two executions of M and M' that yield a different output sequence. The models are equivalent if Formula (4) is unsatisfiable. Checking software equivalence is more complicated, since the output of the models may occur at different times. The algorithmic details of BMC-based equivalence checking are covered in [Kroening *et al.*, 2003]. A predicate-abstraction [Graf and Saïdi, 1997] based approach is presented in [Kroening and Clarke, 2004].

3 Test-Case Generation Using Model Checking

Automatic test-case generation has been studied extensively and a number of different solutions have been proposed (e.g., [Schnurmann *et al.*, 1975; Howden, 1982; DeMillo and Offutt, 1991; Chen *et al.*, 2005]). A detailed discussion of test case generation techniques can also be found in the MOGENTES deliverable D1.2 in Chapter 3.5. Random test case generation still plays a prominent role, since it is easy to implement and to apply. In this section, we briefly discuss the objectives of automatic test case generation; then, we describe how Model Checkers are exploited for the purpose.

We cover test case generation for *reactive systems*, i.e., systems that perform an *ongoing* computation based on *reaction to stimuli* of their environment, and whose output can be observed externally.

3.1 Outline of the Problem

Given an artifact such as a piece of software or a model, we aim at automatically generating a set of test cases by applying a Model Checker to the system under test (SUT). Model Checking is a static analysis technique that requires the availability of a behavioural model of the SUT (as described in Chapter 2). The objective is to generate a test suite that achieves a predefined structural *coverage* of the model.

3.1.1 Availability of a Model

The *availability of the model* for test case generation entails under which conditions a specific test case generation technique can be applied. By model we mean a (possibly abstract) representation of the system under test; this can be a source code, an automaton or Kripke structure, for instance. The techniques we discuss in this deliverable are independent from the representation of the model. We do not restrict the analysis to any specific modelling language; however, when a Model Checker is used to generate test cases, the model must be in general both abstract (for manageability[3]) and have *precise* semantics.

The test case generation approach we present in Chapter 4 assumes the availability of a behavioural model and aims at achieving a structural coverage of the model. This corresponds to the requirements of **white box testing**. In practise, it is sometimes the case that no behavioural model is available for certain parts of the model. For instance, many of the modules available in Simulink are provided as binaries only. In that case, we have to revert to **black box testing** techniques, unless it is possible to perform a static analysis of the binary (using Model Checking tools for binaries [Lim and Reps, 2008; Kinder *et al.*, 2009]). It may still be possible to automatically derive a model of the system (e.g., using the learning technique proposed in [Peled *et al.*, 2001]). Alternatively, if a specification of the interface of the system is given, random testing may be a viable approach. A detailed discussion of these techniques is outside the scope of this deliverable. We focus on test case generation techniques that are based on Model Checking.

As discussed in Chapter 1, we distinguish two scenarios:

1. The model serves as a specification for the implementation and as an oracle for the generated test cases.
2. The model *is* the implementation, and the engineer acts as an oracle and has to specify the outcomes for the generated test cases.

In both cases, we can apply the same test case generation techniques. The scenarios only differ with respect to the source of the expected outcome of the test cases.

3.1.2 Quality of the Test Suite

The efficiency or quality of a test suite can be quantified using a number of different criteria.

Coverage Criteria. Safety standards (such as the IEC 61508) often require that the test suite achieves a certain structural coverage (e.g., MC/DC). We propose to evaluate the test suites based on the number of *injected faults* and *mutations* it is able to detect. A more detailed description of this approach is given in Chapter 4. Our test case generation approach guarantees that the test suite satisfies this criterion.

[3] For more details on abstract representations of faults refer to Deliverable D3.1.

Size of the Test Suite. Since the execution of test cases may be time consuming (in particular, a test case may require an engineer making manual adjustments to the system under test), we prefer small test suites over large ones. Each test case triggers a certain behaviour of the system. If the test cases exercise largely overlapping parts of the system's behaviour, a larger number of test cases is necessary to achieve the required coverage. Test suites generated by means of Model Checking are particularly sensitive to this problem [Hamon *et al.*, 2004]. The issue can be addressed by explicitly blocking similar counterexamples during the test case generation process.

Cost of Execution. A test case that is less time consuming is preferable over a test case that takes more time to execute, if both test cases achieve the same coverage. In the MOGENTES project, the aim is generating *efficient* test cases. The industrial partner Re:Lab provides a striking example of the value of efficient test cases: the testing of their Graphical User Interface of control units for tractors requires optical technology. In a this framework, the number of test vectors that can be executed in a reasonable amount of time is very restricted. This motivates the focus on techniques that allow the generation of test vectors that can be executed efficiently. Since our test case generation approach relies on BMC, we always find the *shorter* test cases first. Note that there is not necessarily a simple correspondence between the steps of the transition relation and the runtime of the corresponding test case. Depending on the availability of detailed information on the cost of test case execution, we may be able to refine this heuristic. Furthermore, we aim at keeping the test suite small (i.e., reduce the number of test cases necessary to achieve the desired coverage).

Cost of Generation. The time necessary to generate the test suite also has to be taken into account, even if the approach is fully automated. If it is too time consuming to generate a test suite that is optimal with respect to (some of) the criteria listed above, we may revert to searching test cases that are less efficient but easier to generate. The distinction between generation and execution cost is fundamental to the evaluation of techniques. For example, techniques based on unguided random generation are extremely efficient in the generation phase; however, their inherent lack of guarantees on what behaviour of the program is stimulated requires extensive testing, with execution of large test suites.

3.2 Generating Test Cases with Model Checkers

We start with a brief description of how any off-the-shelf model checking tool can be used to generate a test-suite that satisfies coverage requirements such as statement-coverage, condition-coverage, MC/DC, or predicate coverage [Ball, 2005]. As mentioned in Chapter 2, counterexamples reported by model checking tools are witnesses for the *violation* of a given property. On the other hand, test case generation aims at finding a test case *satisfying* a coverage property (such as the reachability of a statement or basic block in the model or the fact that a certain variable or condition should evaluate to a specific value). By negating the given coverage property, we can use a Model Checking tool to obtain a witness that satisfies the original, non-negated property.

This approach is described in [Beyer *et al.*, 2004], where a predicate-abstraction-based model checker is used to generate test vectors for C programs. For a given predicate P, the technique reveals each location in the program where P can hold if the appropriate input values are supplied. For each such location, the Model Checker returns an appropriate test case that results in an execution reaching that location with P being satisfied. By applying this technique for each condition P that occurs in the program, we obtain a test suite that provides condition coverage. Setting P to $true$ yields a test suite that covers every reachable location in the C program, and at the same time highlights possible regions of dead code prior the actual test execution.

The authors leverage the BLAST Model Checker to produce test cases with this approach. The generation is performed by alternating two phases. In the first phase, the Model Checker is asked for a counterexample showing that the location of a certain C source is reachable with a given property. As the BLAST Model Checker supports predicate abstraction and refinement, the authors claim this step is scalable enough to be applicable to real examples up to several thousand lines of code. In the second phase, the counterexample generated to reach the given location is used to produce a concrete test vector. This is done by symbolically executing it on a structure similar to the program's Control Flow Graph. From the trace indicated by the counterexample, a *trace formula* is built that constraints the values of the variables to follow that trace. The variables in the trace formula are then extracted as a sample satisfying assignment, and the input variables are eventually extracted to form the test vector.

The fact that the BLAST Model Checker uses lazy abstraction and *incremental* Model Checking [Henzinger *et al.*, 2004] has a positive impact on its application to test case generation: as it attempts to reuse partial

counterexamples for several refinement steps, the set of test cases produces remains small, thus increasing the density and quality of the generated test suite.

Holzer et al. present a similar BMC based approach [Holzer *et al.*, 2008]. Their implementation FSHELL is based on the CBMC tool [Clarke *et al.*, 2004] of ETH Zurich (see Chapter 2.1). FSHELL is able to generate test suites for C programs. The flexible query language (FQL) enables the specification of different coverage criteria [Holzer *et al.*, 2009]. The language subsumes standard coverage criteria like basic block coverage, multiple condition coverage, but it is also possible to express user-defined criteria (specific to the source code, for example). An FQL expression is of the form

in $prefix$ **cover** $goals$ **passing** $scope$

where $prefix$ restricts the coverage criteria to a particular portion of the program, $goals$ expresses the actual coverage criterion and $scope$ restricts test cases to pass through certain program paths only. For example, consider the FQL expression:

in `/code.c/` **cover** `@line(6),@call(func)` **passing** `@file(code.c)` $\setminus$ `@call(not_implemented)`

This FQL query requires line 6 and all function calls `func` in file `code.c` to be covered. Additionally, the statement `@file(code.c)\call(not_implemented)` requires the test cases not to call `not_implemented`. Holzer et al. show how to generate a set of test goals from an FQL query such that the query is satisfied (by a test suite) if and only if each test goal is satisfied (by a test case). Each test goal is then encoded as a formula TG_j over the states explored by the Model Checking tool:

$$\exists s_0, \ldots, s_k \,.\, I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge TG_j \tag{5}$$

An assignment to the states $s_0, \ldots, s_k$ in Formula (5) represents a test case that covers the test goal TG_j. In order to get a test suite that covers *all* test goals, Formula (5) is solved for each TG_j, eventually yielding a solution to the query. To improve performance, the encoding of TG_j is such that Formula (5) needs to be modified in an *incremental* manner only (i.e., the formula is changed by only *adding* a conjunct to it). This exploits the ability of solvers to reuse learnt facts from previous iterations. Holzer et al. present other optimisations that can, for example, minimise the number of test cases in a test suite.

FSHELL and the test case generator based on BLAST extract test vectors from C programs. The Model Checking based test case generation techniques explained above are by no means restricted to implementation languages like C or Java. Micskei and Majzik use Model Checking to generate test vectors for UML state charts [Micskei and Majzik, 2006]. The formalism of transition relations is general enough to apply to state charts as well. Micskei and Majzik evaluate the efficiency of the Model Checking tools SPIN [Holzmann, 2005], SMV [McMillan, 1993] and UPPAAL [Hessel and Pettersson, 2004] for test case generation. Similar to the tools described above, their approach is to use the model checking tools to obtain witnesses for the reachability of states or events.

4 Fault-Based Test Case Generation

The test case generation techniques described in Chapter 3 enable the generation of test suites that satisfy structural coverage criteria such as condition or statement coverage. The coverage criteria for fault-based testing in the MOGENTES project are based on syntactic and semantic modifications of the model. Given a modification to the model, the aim is to generate a test case that demonstrates the resulting change of the behaviour. Simple structural coverage metrics are not sufficient, since even exhaustive coverage criteria such as MC/DC provide no guarantee that the error resulting from the modification of the model has a visible impact on the behaviour generated by exercising the test suite. Given a model M and its modified counterpart M', equivalence checking (see Chapter 2.2) allows us to find input sequences $s_0.i, \ldots, s_k.i$ for which the models yield (sufficiently) different output sequences $s_0.o, \ldots, s_k.o$ by instantiating Formula (4).

The fault driven test case generation approach is inspired by mutation testing and fault injection:

- **Mutation testing** denotes the method of making (syntactic) modifications to the source code of the implementation. The intention is to evaluate a given test suite based on whether it is able to detect the introduced faults and to aid the generation of additional meaningful test cases.
- **Fault injection.** Fault injection triggers the occurrences of faults in the system under test. The main purpose of this technique is to evaluate the error handling mechanisms of the system.

The common idea underlying both approaches is to make modifications to the system and to run test cases that demonstrate the impact of these changes. Our aim is to handle software mutations and fault injection in a uniform manner: The model-based approach of the MOGENTES project allows us to embed software mutations as well as hardware faults in the model of the system by using a set of *fault models*. A catalogue of mutations and fault models is provided in Deliverable D3.1a. In this section, we focus on describing an automated analysis that allows us to extract "good" test cases from a model containing mutations and failure modes.

4.1 Mutation Testing and Fault Injection

From an abstract point of view, mutations as well as injected faults are simply modifications to the behaviour of the model. In our formalism, such a modification can be modelled by replacing the transition relation R of our model M' with a slightly modified transition relation R' (and possibly a modified condition I' for the initial state). The faults introduced into the system may be either permanent, transient, or intermittent (i.e., repeatedly). The former case can be simply modelled by permanently altering the transition relation R, and applying the resulting relation R' in each step:

$$\underbrace{I'(s_0) \wedge R'(s_0, s_1) \wedge R'(s_1, s_2) \wedge R'(s_2, s_3) \wedge R'(s_3, s_4)}_{\text{permanent fault}} \wedge \ldots$$

To model transient or intermittent faults, we have to take the temporal aspect into account, i.e., the alteration becomes only effective at certain points in time. A typical execution satisfies the following constraint:

$$I(s_0) \wedge R(s_0, s_1) \wedge R(s_1, s_2) \wedge \underbrace{R'(s_2, s_3)}_{\text{intermittent fault}} \wedge R(s_3, s_4) \wedge \ldots$$

This can be achieved by augmenting the state space $\mathcal{S}$ with a timer variable $t \in \mathbb{N}$, yielding a new state space $\mathbb{N} \times \mathcal{S}$. Given a transition relation $R \subseteq \mathcal{S} \times \mathcal{S}$ and a modified transition relation $R' \subseteq \mathcal{S} \times \mathcal{S}$, intermittent faults that occur only at certain points in time $\mathcal{T}$ can then be modelled as

$$R''(\langle t, s_0 \rangle, \langle t+1, s_1 \rangle) := \begin{cases} R'(s_0, s_1) & \text{if } t \in \mathcal{T} \\ R(s_0, s_1) & \text{if } t \notin \mathcal{T} \end{cases} \tag{6}$$

Note that this trick need not be applied if we use BMC, since the progression of time is implicitly represented by unrolling the transition function.

The kinds of faults and mutations introduced into the model are the subject of the MOGENTES Deliverable D3.1a. Mutations are small syntactic changes of the model, whereas simulated hardware faults require semantic changes to the model that reflect physical faults of the system as accurately as possible. Conceptually, however, there is no difference when it comes to their integration into the transition relation: The implementation of faults in the model M requires syntactic changes to M.

Depending on the extent of these modifications, the resulting error may not be immediately observable, i.e., it is not necessarily the case that

$$s_0.i = s'_0.i \wedge R(s_0, s_1) \wedge R'(s'_0, s'_1) \implies s_1.o \neq s'_1.o$$

holds. Even though s_1 differs from s'_1, the outputs $s_1.o$ and $s'_1.o$ may be indistinguishable: the modification of R may not necessarily have an (immediate) impact on the observable behaviour. Intuitively, a test case is "good" if it yields a different outcome for M and M'. In mutation testing, the term *weak mutation testing* refers to the condition that the test cases should cause different program states for the mutant and the original model. In the case that the affected part of the state is not observable, this condition is not sufficient for our purpose. *Strong mutation testing* refers to the case where the error propagates to the output of the model and is caught by an appropriate test case. In dependable systems, this notion may be too strong, since redundant systems may tolerate a certain number of faults. Note that this case can be detected using a *complete* Model Checking technique (see Chapter 2.1.4) or k-induction (Chapter2.1.5). In that case, the test case should establish that the backup mechanism prevents the error from having an observable impact on the system. An appropriate test case can be found by injecting a *stronger* fault [Kupferman *et al.*, 2008]. The least aggressive modification causing a redundant system able to tolerate single faults to fail must simulate two simultaneous faults. A test case that detects this more aggressive modification will fail to uncover the single fault, but *would* yield a different behaviour if the redundancy mechanism failed. It is therefore "good" in the sense that it verifies the error handling mechanisms of the system. In particular, it verifies whether the *implemented* error handling mechanism behaves as specified in the model.

4.2 Generating Test Cases

As mentioned previously, one way of generating test cases that detect a mutation is to find a satisfying assignment to Formula (4). Such a satisfying assignment provides the inputs that yield a different output sequence during the first k steps and, provided the observable behaviours of the two models M and M' are not fully equivalent, such a solution must exist for some k. In general, however, k is bounded and the Model Checker may not be able to find a test case if more than k steps are needed to yield a different output sequence. As observed by Holzer et al., this seems not to be the problem for structural coverage. We expect a similar observation for equivalence checking, although this needs experimental confirmation.

4.2.1 Encoding Combinations of Faults

Assume that the objective is to generate a test suite that detects single faults (or mutations). The naïve approach to create such a test suite is to generate a new model M' for each conceivable fault or mutation and to generate an instance of Formula (4) for each pair of models M and M'. In practise, this approach is very wasteful, since modern satisfiability checkers such as MINISAT [Eén and Sörensson, 2004] are able to solve problem instances *incrementally*. Encoding M' in a way such that faults or mutations can be activated or deactivated by adding constraints to the formula allows the SAT solver to (partially) reuse the information it has already derived.

Therefore, we propose to generate a modified model M' that contains all faults and mutations for which we want to generate test cases. We use the same idea as in Equation (6) and introduce a Boolean flag f for each modification that allows us to activate (deactivate) the respective fault/mutation by setting f to true (false). Assume that R is the transition relation of the original model M and that R'_i is the transition relation of the model M'_i that contains the i^{th} of the n modifications in question. We define the model R_μ as follows:

$$R_\mu(s_0, s_1, f_1, \ldots, f_n) := \begin{cases} R(s_0, s_1) & \text{if } \bigvee_{i=1}^{n} f_i = \mathsf{F} \\ R'_1(s_0, s_1) & \text{if } f_1 = \mathsf{T} \wedge (\bigvee_{=2}^{n} f_i) = \mathsf{F} \\ R'_2(s_0, s_1) & \text{if } f_2 = \mathsf{T} \wedge f_1 = \mathsf{F} \wedge (\bigvee_{=3}^{n} f_i) = \mathsf{F} \\ \quad\vdots & \\ R'_n(s_0, s_1) & \text{if } f_n = \mathsf{T} \wedge \left(\bigvee_{=1}^{n-1} f_i\right) = \mathsf{F} \end{cases} \tag{7}$$

We use M_μ to denote the model with the transition relation R_μ. We can further refine the technique if M comprises a control flow graph that partitions R into local transition relations. Let $\mathcal{L}$ be the set of locations determined by the control flow graph of M, R_l the transition relation for the location l, and let $s.\ell$ denote the location of the state s. Then R is defined as the disjunctive partitioning

$$R(s_0, s_1) := \bigvee_{l \in \mathcal{L}} (s_0.\ell = l \wedge R_l(s_0, s_1)) \ .$$

This formalism allows us to introduce separate sets of modifications of the form (7) for *each* location $l \in \mathcal{L}$, leading to a set of mutation flags $\{f_{i,l} \mid 0 < i \leq n, l \in \mathcal{L}\}$.

Given the resulting transition relation R_μ (as defined in (7)) we can construct an instance of Formula (4). A fault in the modified model M_μ can be triggered by adding a constraint of the form

$$F_j := f_j \wedge \bigwedge_{0<i\leq n, i\neq j} \neg f_i \ . \tag{8}$$

The decision procedure on which the CBMC tool is based on performs bit-level accurate reasoning by transforming the instance of Formula (4) into an equi-satisfiable propositional formula EQ_k in Conjunctive Normal Form[4] (CNF). This formula is then handed over to the satisfiability checker MINISAT [Eén and Sörensson, 2004]. The decision process of MINISAT is incremental, i.e., it allows

- to add additional clauses, and
- to add or remove a constraint F_j of the form described in (8)

without restarting the solver, meaning that the solver can reuse intermediate results if it has to solve similar problem instances.

Let EQ_k denote the CNF of the instance of Equation (4) derived from the models M and M_μ. We can generate a test suite covering all n mutations in question by iteratively computing satisfiable assignments to

$$EQ_k \wedge F_i, \quad i \in \{1, \ldots, n\} \ . \tag{9}$$

If each of the instances of Formula (9) has a solution, we obtain n (not necessarily different) test cases that cover all mutations injected into the model M_μ.

4.2.2 Searching for Combinations of Faults

Assume that one instance $EQ_k \wedge F_c$ of (9) (with $i = c$) is unsatisfiable. This indicates that the injected fault corresponding to f_c (see Formula (7)) does not result in an error that propagates to an observable output within k steps. There are two possible reasons for this phenomenon:

1. The bound k is not sufficiently large to reveal the error.
2. The model contains redundancy and the injected fault does not result in an observable change of its behaviour. We say that the model *tolerates* the fault.

A *complete* Model Checking algorithm can distinguish both cases. The first case can be addressed by simply increasing the bound k. In the second case, the resulting test case can then be used whether the implementation adheres to the specification and tolerates the fault, too. Furthermore, we can force the occurrence of an error by injecting a *stronger* fault (as suggested in [Kupferman *et al.*, 2008]). This can be achieved by combining two or more of the faults in the model M_μ. Our setting provides an easy means to trigger more than one fault. Given a set of fault indices $\mathcal{F}$ that correspond to the selected set of faults, we construct the constraint

$$F_\mathcal{F} := \bigwedge_{i\in\mathcal{F}} f_i \wedge \bigwedge_{i\in(\{1..n\}\setminus\mathcal{F})} \neg f_i \ . \tag{10}$$

By solving the instance $EQ_k \wedge F_\mathcal{F}$, we obtain a test case $TC_\mathcal{F}$ that distinguishes the model M and the respective modified model into which the faults/mutations corresponding to $\mathcal{F}$ have been injected. While $TC_\mathcal{F}$ yields the same output for M and any M'_i that contains only a single fault, it is a "good" test case in the sense that it challenges the error handling mechanisms of the implementation.

Our approach to address the cases 1 and 2 described above suffers from two problems:

- Cases 1 and 2 are only distinguishable by means of an (expensive) complete Model Checking procedure.
- In order to avoid a combinatorial explosion of fault combinations, the set $\mathcal{F}$ has to be chosen cautiously.

[4] A propositional formula in conjunctive normal form is a conjunction of disjunctions of literals, where a literal l is a propositional variable or its negation (e.g., a or $\neg a$). A *clause* is a disjunction of literals.

Both issues can only be tackled using heuristics. In our setting, we do not necessarily require a complete verification algorithm, as we are satisfied with a partial result. In order to avoid scalability problems of complete Model Checking procedures, we plan to apply a complete Model Checker to over-approximations of the model, e.g., to localisation reductions $\hat{M}_\mu$ of M_μ. The reduced model $\hat{M}_\mu$ is typically orders of magnitudes smaller than M_μ, which enables the Model Checker to pass. This is a conservative procedure for any reachability property, which includes our coverage criteria. If the Model Checker determines that $\hat{M}_\mu$ tolerates a given fault, so does M_μ, and we do not need to attempt to generate a covering test case. If not so, the Model Checker provides a counterexample on $\hat{M}_\mu$, which we may attempt to conretize to be satisfied by M_μ.

The second problem (choosing an appropriate combination of faults or a stronger fault) can be addressed by defining a partial order over combinations of faults that orders them with respect to their strength (as suggested in [Kupferman *et al.*, 2008]). Let μ and ν be mutations or faults. According to Definition 1 in [Kupferman *et al.*, 2008], μ is *at least as aggressive* than a ν if for every model M and every specification $\mathcal{S}$, we have that if M_μ adheres to $\mathcal{S}$, then so does M_ν. For instance, let μ be a fault that produces the effect that a certain binary signal changes non-deterministically. This fault is stronger than a fault ν that results in the same signal being permanently 1, since the possible behaviours induced by μ also contain the behaviour resulting from ν. Note furthermore that a combination of two arbitrary faults μ and ν is *not* necessarily stronger than the single fault μ, since the fault effects might cancel each other out. The aggressiveness ordering introduced above results in a lattice where the bottom element is the empty set of mutations and the top element represents the most aggressive mutation (which is possibly a combination of several mutations). The aim is now to find a a combined mutation in the lattice that is strong enough to yield a different output. Since the size of the lattice grows exponentially with the size of $\mathcal{F}$, a good heuristic must be chosen in order to avoid combinatorial explosion. Searching the lattice by beginning from the extremes (top or bottom) of the lattice is certainly not a good strategy for the following reasons:

- The closer a combined mutation is to the top element of the element of the lattice, the less likely it is to occur in reality since this would mean that *all* the mutations occur *simultaneously*.
- The closer a combined mutation is to the bottom element of the lattice, the less likely it will have an impact on the output since the (combined) mutation might be to weak and might not propagate to the output.

This suggests a binary search on the lattice as proposed by Purandare et al. [Purandare *et al.*, 2009]. That is, the search starts from the middle of the lattice. In case a combined mutation is found not to propagate to the output, all weaker mutations can be deleted form the lattice, since a weaker mutation will not affect the output either. In such a case the search tries to find a stronger mutation. Otherwise, the algorithm outputs the current combined mutation or tries to find a weaker one. Purandare et al. [Purandare *et al.*, 2009] apply a similar technique for checking whether specifications vacuously hold. This work is, as observed by Kupferman et al. [Kupferman *et al.*, 2008], closely related to our problem; the problem of finding mutations that violate a specification is dual to the problem of vacuity checking.

4.2.3 Generating Test Cases for Continuous Models

Another issue is the presence of floating-point computations in the models M and M'. Currently, no Model Checker is able to handle such computations accurately, because floating-point operations are usually modelled using arithmetic over the reals. This approximation does not take the limited accuracy of floating point numbers into account. In particular, using such an approach, modelling the bit-level faults required by the MOGENTES demonstrators (a catalogue is provided in Deliverable D3.1a) is extremely hard.

A Model Checker supporting floating-point operations is, however, highly desirable since floating-point operations often do not behave as programmers expect. For example, associativity does not hold for floating-point operations, i.e., $(a \oplus b) \oplus c$ may not be equal to $a \oplus (b \oplus c)$, where $\oplus$ is floating-point addition. Test cases generated by such a Model Checker can thus reveal unexpected results. We aim at extending state-of-the-art Model Checkers such as CBMC to generate test cases for models that have floating-point numbers. Next, we provide a description of such a Model Checker.

A Model Checker that supports floating-point operations can be realised as follows. Given a circuit implementation of floating-point unit (FPU), each floating-point operation is described with a formula in bit-vector arithmetic. These formulas are supported by state-of-the-art Model Checkers and can be incorporated to the description of the program for which we would like to generate test cases. This, however, yields difficult SAT instances. We discuss ways to obtain instances with better complexity.

Using Approximations to Generate Test Cases

Experimental results show that SAT instances are more expensive to decide if

- the precision of the floating-point operations is increased. For example, it is considerably more difficult to generate test cases for a program with double precision than for the same program with single precision.
- the range of the exponent is increased. This is due to *alignment* step required for floating-point addition/ subtraction, i.e., the smaller the exponent range, the less complex is the alignment and the less difficult is the SAT instance.

Considering these two empirical results, we propose to generate *approximations* of floating-point programs that are sufficient to generate valid test cases but are simpler for the SAT solver. More precisely, we will alternate between *over-approximations* and *under-approximations* of the program containing floating-point operations. An over-approximation is a program with *more* execution traces than the unapproximated version. Conversely, an under-approximation is a program with *less* execution traces than the unapproximated version. Note that only under-approximations always provide valid test cases since no "wrong" traces are added. However, over-approximations are useful to guide the construction of good under-approximations, i.e., approximations that are more likely to generate a test case faster.

We will reduce the precision to generate an over-approximation and we will reduce the range of the exponent to obtain an under-approximation. Algorithm 1 shows how to alternate between over-approximations

Algorithm 1 Generate test cases for programs containing floating-point operations

Input: $\mathcal{P}$: Program containing floating-point operations
Output: t: a test case (empty test case if none)

1: **let** $\overline{\mathcal{P}}$ be an over-approximation of $\mathcal{P}$ obtained by reducing the precision in $\mathcal{P}$.
2: **let** $\widehat{\mathcal{P}} := \overline{\mathcal{P}}$
3: **loop**
4: run CBMC ($\widehat{\mathcal{P}}$)
5: **if** SAT $\wedge$ $\widehat{\mathcal{P}}$ is over-approx **then**
6: generate an under-approximation $\underline{\mathcal{P}}$ by reducing the range
7: refine $\underline{\mathcal{P}}$ by using the returned trace
8: $\widehat{\mathcal{P}} := \underline{\mathcal{P}}$
9: **else if** UNSAT $\wedge$ $\widehat{\mathcal{P}}$ is under-approx **then**
10: generate an over-approximation $\overline{\mathcal{P}}$ by reducing the precision
11: refine $\overline{\mathcal{P}}$ using the unsatisfiable core
12: $\widehat{\mathcal{P}} := \overline{\mathcal{P}}$
13: **else if** UNSAT $\wedge$ $\widehat{\mathcal{P}}$ is over-approx **then**
14: **return** empty test case t // there is no execution trace
15: **else if** SAT $\wedge$ $\widehat{\mathcal{P}}$ is under-approx **then**
16: **return** test case t from satisfying assignment

and under-approximations. It is based on the Model Checker CBMC, which is able to return traces in cases there exists one (SAT case). Otherwise, CBMC returns UNSAT. Details on how to generate the approximations and how to refine them is out of the scope of this deliverable, since it requires an in-depth discussion of floating-point computations. We provide details in an upcoming publication.

A New Notion of Equivalence

In the context of fault-based test-case generation, we are also exploring the possibility of relaxing the notion of equivalence. More precisely, in equation 4, if $s_i.o$ and $s'_i.o$ are floating-point outputs, we will replace the constraint

$$\bigvee_{i=0}^{k} s_i.o \neq s'_i.o \text{ in equation 4 by the constraint } \bigvee_{i=0}^{k} |s_i.o - s'_i.o| \geq \delta_i \ .$$

That is, instead of requiring a test case to result in a different output, we require the output to violate a tolerance bound given by δ_i. This is reasonable if small deviations in the output are acceptable. Such a relaxation of the notion of equivalence could also speed up the generation of test cases.

Since many fault models listed in Deliverable D3.1a are based on bit-level modifications (such as *single-bit-stuck-at* faults), there is a strong case for the SAT-based, bit-level accurate decision procedures we apply. In a

propositional encoding, single-bit modifications can be integrated very easily, and SAT-solvers can deal with the resulting encoding very efficiently. For instance, modelling a single-bit-stuck-at-1 fault corresponds to setting a single propositional variable in the formula EQ_k to true. Modern SAT-solving algorithms deal with this case by using unit propagation, which is extremely efficient. If, on the other hand, we would use a decision procedure based on the popular Simplex algorithm (an algorithm for solving linear arithmetic equations), encoding the same fault would be complicated and would result in an equation that is very hard to solve.

4.2.4 Generating Test Cases for Stress Testing

The generation of *stress tests* is another goal of the MOGENTES project. Conceptually, the test case generation approach presented in Chapter 3.2, Formula(5), can be applied. The challenge is to encode the concept of a test case that stresses the system under test in the constraint TG_j. Furthermore, the resulting decision problem is an *optimisation* problem, since we intend to extract the test case that puts the most stress on the system.

For instance, the objective could be to maximise the number of a certain kind of events.[5] Given a bound k, we want to generate a test case in which as many instances of an event e as possible occur. Assume that we are given a predicate $E(s_i)$ that evaluates to 1 if the event e occurs in the state s_i, and to 0 if it does not. We can construct an expression that indicates the number of occurrences of the event e in the execution $s_0, \ldots, s_k$. We can then generate a stress test by solving the following problem:

$$\text{maximise } b_0 + \ldots + b_k \text{ such that } \exists s_0, \ldots, s_k \,.\, I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k} E(s_i) = b_i \tag{11}$$

We can either try to compute an accurate solution to this problem using a *pseudo-Boolean solver* such as PBS[6], or we can use incremental SAT-solving techniques combined with an eager approach that uses a heuristic to set as many of the variables $b_0, \ldots, b_k$ in (11) to 1. A similar technique has been used to improve the quality of counterexamples in [Groce and Kroening, 2005], where we propose to *minimise* the absolute value of variables that occur in a counterexample in order to obtain traces that are easier to understand.

Given an (approximated) solution to (11), we can use the techniques described in Chapter 3.2 to extract an appropriate test cases that puts the system under stress.

For instance, given a fixed topology for a railway network, a test that sets the system under stress can be a scenario that results in a lot of admissibility checks in a short time span when a route is set or resolved. In this case, the bound k is defined by the time span that is specified, and $E(s_i)$ evaluates to 1 if an admissibility check is performed.

4.3 Issues Specific to the Modelling Language

Simulink[7] has become the de-facto standard for modelling in many application domains, such as automotive control. It provides a graphical modelling language that allows to create high-level models efficiently and to simulate them to monitor correctness and performance. Simulink models can also be used to generate production level code using commercial code-generation tools such as the Real-time Workshop[8] by MathWorks or TargetLink by DSPACE[9]. Stateflow[10] further enhances Simulink by providing features to develop state machines and flow charts.

A reasonable amount of commercial tools exist for test case generation from Simulink models. Some prominent examples of such tools are Reactis by Reactive Systems [Systems, 2006], Simulink V&V [MathWorks, 2006] and Simulink Design Verifier [MathWorks, 2007] by MathWorks Inc., Simulink Tester by T-VEC [T-VEC, 2008], Safety Test Builder [Software, 2008] by TNI-Software, BEACON Tester [Dynamics, 2007] and AutoMOTGen [Gadkari *et al.*, 2008]. However, most of these tools are have limitations with respect to model size, model structure, or the type of the model. A more detailed discussion of test case generation tools is provided in the MOGENTES Deliverable D1.2.

[5] The term "event" may also refer to internal actions triggered by a test case, e.g., the number of checks the system has to perform to guarantee that a certain operation is safe.

[6] http://www.eecs.umich.edu/~faloul/Tools/pbs/

[7] http://www.mathworks.com/products/simulink

[8] www.mathworks.com/products/rtw/

[9] www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/targetli.cfm

[10] http://www.mathworks.com/products/stateflow/

As mentioned in Chapter 3.1, it is important that a model has precise semantics for test case generation using a Model Checker. Simulink, however, was developed mainly as a simulation environment, and lacks precise semantics:

- The semantics of Simulink depend on the user-configured options of the model, and thus a particular model may behave differently depending on the options selected [Caspi *et al.*, 2003].
- The code generated from these models (using commercially available code generators) does not preserve the semantics of the simulator and may prioritise memory and performance optimisations [Caspi *et al.*, 2003].
- Simulink models may also be extended by writing S-functions. S-functions are basically an implementation of a Simulink block provided in a language such as Matlab, C, C++, Ada, Fortran, or even in terms of a binary. This introduces the problem that parts of the model may not be available in a language that the model checking tool can handle.

The last problem can be addressed by extending the static analyser to be able to deal with more input languages (e.g., binary code [Holzer *et al.*, 2009]); however, only a limited number of languages can be supported in practise. In this case, it may be necessary to apply black box testing techniques such as random testing (as discussed in Chapter 3.1).

In addition to the lack of precise semantics, Simulink does not perform strong static checks (such as type checking [Alur *et al.*, 2008]). Types are supported in Simulink, but are not required to be declared explicitly. Further complications arise when we consider Simulink clocks. In Simulink, the clock semantics is continuous, meaning that all discrete signals are actually piece-wise constant continuous signals [Caspi *et al.*, 2003]. Each signal can have an associated sample time, which is derived from the block producing the signal. These blocks can have different clocks. Another timing mechanism in Simulink is through triggers. Triggers are only applicable to sub-systems in Simulink. Blocks within a triggered sub-system inherit their timing from the sample time of the triggering signal. Furthermore, there are enabled sub-systems in which a block within a sub-system can have a different clock from the parent system. All these clock variants introduce different classes of timing faults and complicate test-case generation from Simulink models.

To address these problems and to achieve TCG using a Model Checker, the first goal is to translate Simulink models into the input format of CBMC. Initially, we restrict our models to only the discrete library of Simulink, starting with a unique clock throughout the system. (The Steering Anti-Catch-Up – or SAC, for short – demonstrator provided by Ford uses the same clock for all blocks.) Furthermore, we will assume the existence of a behavioural model for all components of the system, meaning that the initial version of the test case generation tool does not support S-functions. Once we have accomplished these initial goals, we will move towards enhancing the features supported by our tool chain.

5 Abbreviations and Definitions

Abbreviation	Explanation
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
CBMC	C Bounded Model Checker
CFG	Control Flow Graph
DSP	Digital Signal Processor
EBMC	Extended Bounded Model Checker
FPU	Floating Point Unit
MC	Model Checking
MC/DC	Modified Condition/Decision Coverage
MDA	Model Driven Architecture
MDD	Model Driven Development
MUT	Model under test
SAC	Steering Anti-Catch-Up
SAT	Satisfiability
SUT	System under test
TCG	Test case generation
UNSAT	Unsatisfiable
WCET	Worst Case Execution Time

References

[Alpern *et al.*, 1988] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages (POPL)*, pages 1–11. ACM, 1988.

[Alur *et al.*, 2008] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. *Embedded Software (EMSOFT)*, October 2008.

[Ball, 2005] Thomas Ball. A theory of predicate-complete test coverage and generation. In *Formal Methods for Components and Objects (FMCO)*, volume 3657 of *LNCS*, pages 1–22. Springer, 2005.

[Barrett *et al.*, 1998] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Design Automation Conference (DAC)*, pages 522–527. ACM, June 1998.

[Berezin *et al.*, 2005] Sergey Berezin, Vijay Ganesh, and David Dill. A decision procedure for fixed-width bit-vectors. Technical report, Computer Science Department, Stanford University, 2005.

[Beyer *et al.*, 2004] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *International Conference on Software Engineering (ICSE)*, pages 326–335, 2004.

[Biere *et al.*, 1999] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[Bryant *et al.*, 2007] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*. Springer, 2007.

[Cadar *et al.*, 2006] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Computer and Communications Security (CCS)*, pages 322–335. ACM, 2006.

[Caspi *et al.*, 2003] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis. Translating discrete-time Simulink to Lustre. In R. Alur and I. Lee, editors, *Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 84–99. Springer, 2003.

[Chen *et al.*, 2005] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. *Advances in Computer Science - ASIAN 2004*, pages 320–329, 2005.

[Clarke *et al.*, 1999] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.

[Clarke *et al.*, 2004] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[Cook *et al.*, 2005] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 296–300. Springer, 2005.

[Currie *et al.*, 2000] David W. Currie, Alan J. Hu, and Sreeranga P. Rajan. Automatic formal verification of DSP software. In *Design Automation Conference (DAC)*, pages 130–135. ACM, 2000.

[Cyrluk *et al.*, 1997] David Cyrluk, M. Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Computer Aided Verification (CAV)*, LNCS, pages 60–71. Springer, 1997.

[DeMillo and Offutt, 1991] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering (TSE)*, 17(9):900–910, 1991.

[Dutertre and de Moura, 2006] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Available at `http://yices.csl.sri.com/tool-paper.pdf`, September 2006.

[Dynamics, 2007] Applied Dynamics. BEACON for Simulink. `http://www.adi.com/pdfs/product/BEACON4S_DS3.pdf`, 2007.

[Eén and Sörensson, 2004] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[Gadkari *et al.*, 2008] Ambar Gadkari, Anand Yeolekar, J. Suresh, S. Ramesh, Swarup Mohalik, and K. C. Shashidar. AutoMOTGen: Automatic model oriented test generator for embedded control systems. In A. Gupta and S. Malik, editors, *Computer Aided Verification*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 204–208. Springer Berlin/Heidelberg, 2008.

[Godlin, 2008] Benny Godlin. Regression verification: Theoretical and implementation aspects. Master's thesis, Technion – Israel Institute of Technology, 2008.

[Graf and Saïdi, 1997] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[Groce and Kroening, 2005] Alex Groce and Daniel Kroening. Making the most of BMC counterexamples. In *Bounded Model Checking (BMC)*, volume 119 of *ENTCS*, pages 67–81. Elsevier, 2005.

[Hamon *et al.*, 2004] Gr Egoire Hamon, Leonardo De Moura, and John Rushby. Generating efficient test sets with a model checker. In *In: 2nd International Conference on Software Engineering and Formal Methods*, pages 261–270. IEEE Press, 2004.

[Henzinger *et al.*, 2004] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. Extreme model checking. *Verification: Theory and Practice*, pages 180–181, 2004.

[Hessel and Pettersson, 2004] Anders Hessel and Paul Pettersson. A test case generation algorithm for real-time systems. In *Quality Software International Conference (QSIC)*, pages 268–273. IEEE, 2004.

[Holzer *et al.*, 2008] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. FShell: Systematic test case generation for dynamic analysis and measurement. In *Computer Aided Verification (CAV)*, volume 5123 of *LNCS*, pages 209–213. Springer, 2008.

[Holzer *et al.*, 2009] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Query-driven program testing. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, LNCS. Springer, 2009. To appear.

[Holzmann, 2005] Gerard J. Holzmann. Software model checking with SPIN. *Advances in Computers*, 65:78–109, 2005.

[Howden, 1982] W.E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering (TSE)*, SE-8(4):371–379, July 1982.

[Kinder *et al.*, 2009] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[Kroening and Clarke, 2004] Daniel Kroening and Edmund Clarke. Checking consistency of C and Verilog using predicate abstraction and induction. In *IEEE/ACM International conference on Computer-aided design*, pages 66–72. IEEE, 2004.

[Kroening and Strichman, 2003] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2575 of *LNCS*, pages 298–309. Springer, 2003.

[Kroening *et al.*, 2003] Daniel Kroening, Edmund M. Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference (DAC)*, pages 368–371. ACM, 2003.

[Kuehlmann and van Eijk, 2002] Andreas Kuehlmann and Cornelis A. J. van Eijk. Combinational and sequential equivalence checking. In *Logic Synthesis and Verification*, Kluwer International Series In Engineering And Computer Science Series, pages 343–372. Kluwer, Norwell, MA, USA, 2002.

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

[Lim and Reps, 2008] Junghee Lim and Thomas W. Reps. A system for generating static analyzers for machine instructions. In *Compiler Construction (CC)*, volume 4959 of *LNCS*, pages 36–52. Springer, 2008.

[MathWorks, 2006] MathWorks. Simulink verification and validation 2. `https://tagteamdbserver.mathworks.com/ttserverroot/Download/35746_91211v02_SL_VV_ds_web.pdf`, September 2006.

[MathWorks, 2007] MathWorks. Simulink design verifier 1. `https://tagteamdbserver.mathworks.com/ttserverroot/Download/40880_91458V00_sldesignverifier_web.pdf`, May 2007.

[McMillan, 1993] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[Micskei and Majzik, 2006] Zoltán Micskei and István Majzik. Model-based automatic test generation for event-driven embedded systems using model checkers. In *Dependability of Computer Systems (DepCoS-RELCOMEX)*, pages 191–198. IEEE, 2006.

[Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535. ACM, 2001.

[Peled *et al.*, 2001] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2001.

[Purandare *et al.*, 2009] Mitra Purandare, Thomas Wahl, and Daniel Kroening. Strengthening properties using abstraction refinement. In *Design, Automation, and Test in Europe (DATE)*. ACM, 2009. To appear.

[Schnurmann *et al.*, 1975] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter. The weighted random test-pattern generator. *IEEE Transactions on Computers*, 24(7):695–700, 1975.

[Sheeran *et al.*, 2000] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

[Software, 2008] TNI Software. Safety test builder: Product overview. `http://www.tni-software.com/commun/docs/safetytestbuilder.pdf`, 2008.

[Systems, 2006] Reactive Systems. *Model-Based Testing and Validation with Reactis*. `http://www.reactive-systems.com/papers/bcsf.pdf`, 2006.

[T-VEC, 2008] T-VEC. Automated test generation and execution for Simulink. `http://www.t-vec.com`, October 2008.

[Wedler *et al.*, 2005] Markus Wedler, Dominik Stoffel, and Wolfgang Kunz. Normalization at the arithmetic bit level. In *Design Automation Conference (DAC)*, pages 457–462. ACM, 2005.

[Yang *et al.*, 2006] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson R. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy (S&P)*, pages 243–257. IEEE, 2006.