Mutation-based Test Case Generation for Simulink Models

Angelo Brillout[1], Nannan He[2], Michele Mazzucchi[1], Daniel Kroening[2], Mitra Purandare[1], Philipp Rümmer[2], and Georg Weissenbacher[1,2]

[1] Computer Systems Institute, ETH Zurich
[2] Computing Laboratory, Oxford University

Abstract. The Matlab/Simulink language has become the standard formalism for modeling and implementing control software in areas like avionics, automotive, railway, and process automation. Such software is often safety critical, and bugs have potentially disastrous consequences for people and material involved. We define a verification methodology to assess the correctness of Simulink programs by means of automated test-case generation. In the style of fault- and mutation-based testing, the coverage of a Simulink program by a test suite is defined in terms of the detection of injected faults. Using bounded model checking techniques, we are able to effectively and automatically compute test suites for given fault models. Several optimisations are discussed to make the approach practical for realistic Simulink programs and fault models, and to obtain accurate coverage measures.

1 Introduction

Model-based design is a development methodology for modern software artifacts. It promotes the use of powerful and specialized modeling languages, allowing the engineer to focus on the domain-specific aspects of the system under development. The implementation of the system is either generated or derived manually from high-level models. The goal is to identify design flaws as early as possible in the development cycle, thereby avoiding costly late-stage design fixes.

The Matlab/Simulink language, developed by The MathWorks,[3] has emerged as the predominant modeling formalism in the automotive industry and is also widely deployed for avionic applications. A software glitch in these application domains may result in high cost and considerable damage of reputation. Due to the safety-critical nature of these domains, defects in the software may put human lives at stake. Accordingly, international safety standards such as DO-178B or IEC 61508 demand the application of rigorous verification techniques. In particular, they require the test engineers to provide a set of test cases that exercise the implementation of the system according to certain coverage metrics. The

Supported by the EU FP7 STREP MOGENTES (project ID ICT-216679) and the ARTEMIS CESAR project.

[3] `http://www.mathworks.com/products/simulink/`

effort to create appropriate test suites is substantial, and the execution of the test suites is time consuming. There is therefore a strong incentive to automate the generation of test cases and to keep the resulting test suite small.

Model-based testing is an application of model-based design for deriving test cases from a model of the design, which promises better scalability and is applicable before the implementation phase. In the context of model-based testing, the question of suitable coverage metrics has to be reconsidered: traditional metrics such as location or branch coverage are no longer meaningful, since the test cases are not derived from implementation source code. In this paper, we focus on *mutation testing*: the quality of the test suite is assessed by injecting mutations into the model and by measuring which percentage of these modifications can be detected when exercising the test cases. We say that a modification is detected if we can observe that the modified and the original model generate different output signals. The resulting test vectors can be used to check that the model satisfies requirements or can be applied to an implementation of the design.

The generation of test suites that achieve high mutation coverage is difficult. We use *model checking* [1] for this task, as it is also able to address the issue of *equivalent mutants*. The application of model checking to generate high-coverage test suites has become commonplace (see, for instance, [2] for an application in the automotive domain). Test case generation for Simulink models is complicated by the fact that the Simulink language lacks a formal semantics and makes heavy use of floating-point arithmetic.

Contribution. We describe an application of the bounded model checking engine CBMC [3] to generate test suites for Simulink models with high mutation coverage. The implementation features precise reasoning with respect to the floating-point semantics of the models. The computational complexity of the underlying model checking algorithm requires us to deploy a number of heuristics to achieve the desired coverage if the number of mutations is large. Moreover, these heuristics also serve the orthogonal purpose of keeping the number of redundant test cases small in order to reduce the time required to execute the test suite.

Related Work. We briefly relate our work to other tools that generate test-vectors by means of software model checkers. A number of papers report applications of CBMC or similar techniques for generating high-coverage test suites [4–6]. These implementations are very similar to ours. There are also reports of the use of predicate abstraction in test-vector generation, e.g., using SLAM [7] and BLAST [8].

We refer the reader to [9] for a broad survey on mutation testing. We only consider mutant models with single mutations, whereas other authors also consider combinations of faults [10]. Do and Rothermel [11] proposes to use mutations to prioritise test cases to increase a test suite's rate of fault detection.

Schuler et al. [12] discusses the impact of *equivalent mutations* (mutations that keep the semantics of the model unchanged) and presents an approach to detect such mutations by means of checking dynamic invariants. We propose a

similar approach in Section 5, but we rely on invariants statically generated by means of verification techniques such as k-induction.

We also relate our work to other methods for analyzing Simulink models. Most tools that aim at formal analyses of Simulink models focus on a particular and usually relatively small fragment. In particular, models that contain ANSI-C are often not considered [13–15]. Strichman and Ryabtsev [16] uses an automated decision procedure to validate code generated by Simulink against a set of verification conditions extracted from the model.

Another issue is the floating-point semantics of Simulink. Tools such as the Simulink Design Verifier rely on approximations of floating point arithmetic by means of infinite-precision rational numbers [17]. In contrast to that, we use a bit-accurate representation of floating-point arithmetic, as presented in [18]. We are therefore able to analyse the exact behaviour of the model rather than an approximation. Furthermore, our bit-level technique enables the use of mutations such as bit-flips in data values.

Outline. Section 2 describes the transformation of Simulink diagrams into an intermediate representation amenable to static analysis. This translation process conclusively determines the semantics of the model. Section 3 discusses how fault injection and mutations of the model and (bounded) model checking can be used to generate test cases. For this purpose, we rely on a model checking technique able to deal with floating-point arithmetic. We present a novel algorithm which aims at identifying efficient test cases that cover more than one mutation, thus reducing the size of the test suite and improving the performance of the test-case generation process in Section 4. In Section 5, we discuss strategies to detect mutations that do not have an observable impact on the model.

2 Simulink Models

The Simulink language is a graphical modeling language comprising block diagrams and an extensive set of block libraries. An example of a Simulink model is presented in Figure 1. Due to the complexity of the language and the lack of formal semantics, Simulink models are not directly amenable to automated analysis. We present a front-end to transform these models into intermediate ANSI-C programs with well-defined semantics. This transformation is performed fully automatically and allows us to separate the ambiguity issues in the Simulink semantics from our test case generation process.

Each Simulink block is associated with a type. We define the meaning of blocks of a particular type by means of C code, which is organized in a library. This library can be independently refined or modified (e.g., to add a new block-type definition) without the need to change the transformation process.

A *block-type* `X` is defined with the following artifacts:

- `X_in_t`: a C struct that defines the input ports of `X`.
- `X_out_t`: a C struct that defines the output ports of `X`.
- `X_props_t`: a C struct that specifies the verification-relevant properties of `X`.

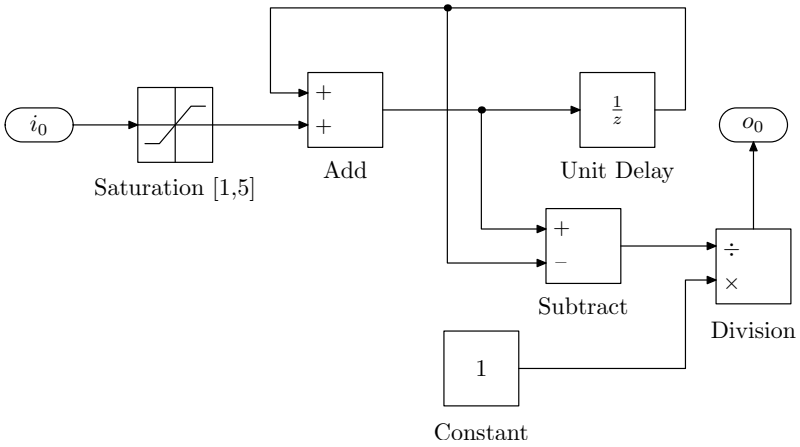


Fig. 1: A Simulink model example

– `X_semfun`: a C function that defines the semantics of `X` as a mapping from input ports to output ports. Its declaration format is `X_out_t* X_semfun(X_props_t *prop, X_in_t *in)`.

Currently, our Simulink front-end supports a set of commonly used block types that belong to a variety of Simulink libraries, such as *Math, Discrete, Logic and Bit, Sinks, Sources, Ports&Subsystems, Discontinuities, Signal Routing, Signal Attributes* and *Model Verification*. Some important block types include *Subsystems*, *Unit Delay* that forms feedback loops, *Switches*, and *From/Goto* pairs. With our uniform definition artifacts, further block types can easily be added to the library as needed. Support for Stateflow diagrams is under development.

The transformation source is a Simulink model composed of a set of interconnected blocks, organized into a hierarchy of subsystems. The front-end parses the model, flattens the hierarchy and derives an appropriate block execution order which determines the simulation order of blocks in the resulting ANSI-C model. The front-end identifies the type and property specifications of each individual block instance and maps it to the corresponding block-type definition in the library. Then, it determines which C block-type definitions need to be included and assigns values to block properties. Some analyses like type inference are also performed during the transformation to resolve ambiguities in the Simulink model.

Figure 2 shows the basic structure of the C code transformed from the model in Figure 1 with the front-end. Our analysis and test-case generation approach based on the resulting C code is introduced in the sections below.

3 Generating Test Cases using Mutations

3.1 Overview

Many existing test case generation techniques permit the generation of test suites that satisfy structural coverage criteria such as condition or statement coverage.

```
/*1.Links to blocktypes definitions */
#include <Sum.h>    // Corresponds to the Add block
...

/*2.Declaration and initialization of block instances */
Sum_in_t b3_in;
Sum_out_t b3_out;
const Sum_prop_t b3_props={.Inputs="++"}; ...

/*3.Simulation loop: define data dependencies w.r.t.
   block connections extracted from the model */
int main() {
  for(sim_time=START; sim_time<END; sim_time+=sim_step) {
    b3_in.port1 = b7_out.port1;
    b3_in.port2 = b2_out.port2;
    b3_out = *Sum_semfun(&b3_props, &b3_in); ...
  }
}
```

Fig. 2: Structural overview of the C code transformed from a Simulink model

One approach to achieve such coverage is to use a *model checker*, which can generate counterexamples that demonstrate the reachability of certain statements or conditions.

The following sections present mutation-based test case generation (TCG) using *bounded model checking* (see Section 3.2). We describe how test cases can be extracted automatically from a model or implementation M by injecting mutations or faults into M (producing in a *mutant model* M') and checking the equivalence of M and M' by means of model checking. If M and M' are not equivalent, a model checking tool is able to generate a witness for the inequality (a counterexample for equality). This counterexample determines a set of input values for which the executions of M and M' produce different outputs.

The coverage criteria for fault-based testing in our work 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 behavior. Simple structural coverage metrics are not sufficient, since even exhaustive coverage criteria such as modified condition/decision coverage (MC/DC) provide no guarantee that the error resulting from the modification of the model has a visible impact on the behavior generated by exercising the test suite.

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.

Examples for injected faults and mutations are provided in Section 3.5. 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.

The following subsection provides a brief overview over the formal verification techniques we apply to generate test suites from models containing mutations or failure modes.

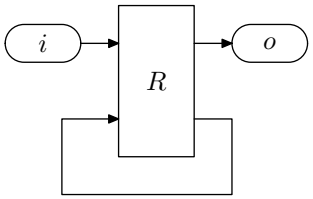
3.2 Bounded Model Checking

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 [1]. It differs from testing in so far as it aims at an exhaustive exploration of the state space of the model or program under test, thereby 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 that demonstrates how the specification can be violated in the model.

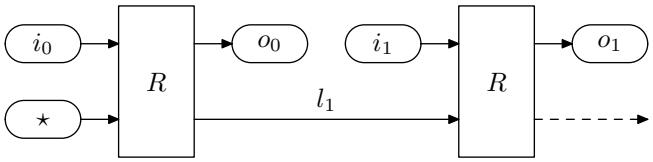
Bounded model checking (BMC) is a variation of model checking which restricts the exploration to execution traces up to a certain (user-defined) length k. BMC either provides a guarantee that the first k execution steps of the program are correct with respect to the property P or a counterexample of length at most k. The ability to report counterexamples is the essential feature we use to generate test cases. The disadvantage of model checking is that it does not scale as well as testing.

Figure 3a illustrates the schema of a Simulink model with a feedback loop. Simulink diagrams comprise blocks instances and signals and wires representing the connections between these blocks. These components determine the input and output signals (i and o, respectively) and the transition function R represented by the model. A bounded model checking algorithm unwinds such models as indicated in Figure 3b; the signals i_i and o_i refer to the input and output signals in the i^{th} step (or point in time), respectively, and $\star$ denotes an undefined/non-deterministic signal value.

For the purpose of test case generation it suffices to determine whether certain states in a model are reachable. A model is specified by a formula representing a (possibly partial) transition relation R (e.g., specified by means of a Simulink diagram or ANSI-C program) 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*, which partitions R into a separate transition function for each program location. This simple formalism is sufficiently general to allow imperative models (such as C programs or state charts) as well as data flow models (such as Simulink models).



(a) Schema of a Simulink model with a loop



(b) Unwinding of the model in Figure 3a

Fig. 3: Unwinding Simulink models

Furthermore, the predicate I characterizes 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, ..., 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 and signals) 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.

3.3 Equivalence Checking

Formal Equivalence Checking is a technique used to formally prove that two models M and M' exhibit the same observable behavior [19]. This is achieved by comparing the input and the output behavior of the two models. To construct a test-scenario (i.e., a sequence of input/output pairs), we are interested in checking whether for a given input sequence the outputs in the first k steps of the executions of the models match. This notion of "k-equivalence" is decidable, assuming that the input and output values have a finite range.

Whether two given models are k-equivalent can be decided using model checking. 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

that the size of the states is finite) whether the following is satisfiable:

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

Any satisfying assignment to this formula represents two executions of M and M' that yield a different output sequence. The models are equivalent (up to k steps) if Formula (1) is unsatisfiable.

Checking software equivalence is more complicated, since the two programs rarely perform input/output in lockstep. Algorithmic details of equivalence checking using BMC are covered in [3]. An approach based on predicate abstraction [20] is presented in [21].

3.4 Floating-Point Arithmetic

Simulink models make heavy use of floating-point arithmetic (FPA). Although reasoning about FPA is an active field of research, existing methods are primarily tailored to interactive proof assistants (e.g., [22, 23]) or abstract floating-point number to intervals of the reals (e.g., [24, 25]). The methods of the first kind are unsuitable for automated tools, while the latter ones are not able to construct models for satisfiable formulae and therefore cannot be used for test-case generation. Currently, there are only few model checkers that handle FPA accurately.

In our model checker CBMC, we use the mixed abstraction framework described in [18]. By using both over- and under-approximations simultaneously, combined with a novel abstraction refinement approach, we are able to achieve both accurate reasoning and a significantly better performance than ordinary bit-blasting approaches [26].

Previous work on test case generation for Simulink programs, for instance in the Simulink Design Verifier [17], employs approximations of floating-point arithmetic by means of infinite-precision rational numbers. While this allows efficient reasoning in the case of models only containing linear arithmetic, rationals do not faithfully reflect the actual behavior of Simulink programs: in equivalence checking, it can happen that Simulink models are erroneously reported as equivalent (although they are not with FPA semantics), or that equivalent Simulink models are erroneously reported as non-equivalent. This is particularly relevant because unexpected over- or underflows due to FPA semantics can be hard to detect, but can have profound consequences.

We illustrate the inconsistencies between rational arithmetic and FPA using the Simulink model in Figure 1. The model contains a feedback loop that

computes the consecutive sums

$$t_n = \sum_{j=1}^{n} i_j$$

given the stream $i_1, i_2, i_3, \ldots$ of inputs. Furthermore, in each time frame, the quotient

$$\frac{1}{t_n - t_{n-1}}$$

is computed. Because the inputs are constrained to the interval $[1, 5]$ with the help of a `Saturation` block, and since $t_n = t_{n-1} + i_n$, the computation of the fraction will always succeed when computing with infinite-precision rational numbers. If the model is implemented and executed using floating-point numbers, however, the stream of sums $t_1, t_2, t_3, \ldots$ will eventually get stationary due to lack of precision: in FPA, it is the case that $a + b = a$ if a is a very large and b a very small number. As soon as $t_n = t_{n-1}$ occurs in the sequence of sums, the computation of the quotient will raise a division-by-zero exception.

The use of a bit-accurate decision procedure has further advantages in the context of mutation testing: Many fault models are based on bit-level modifications (such as *single-bit-stuck-at* faults). The effect of these mutations is trivial to model using a propositional formula, and SAT-solvers can deal with the resulting encoding very efficiently. For instance, modeling a single-bit-stuck-at-1 fault corresponds to setting a single propositional variable to true. Modern SAT-solving algorithms deal with this case by using unit propagation, which is extremely efficient. If, on the other hand, a decision procedure for reasoning about real arithmetic was used, encoding the same fault would be complicated and would result in constraints that are very hard to solve.

3.5 Mutation Testing and Fault Injection

From an abstract point of view, mutations as well as injected faults are simply modifications to the behavior of the model.

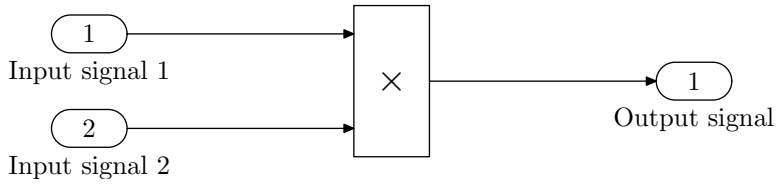
Example 1. Consider the simple Simulink diagram in Figure 4a. The input signals i_1 and i_2 are related to the output signal o by means of the formula $o = i_1 \times i_2$, i.e., the transition function is

$$R(s_i, s_{i+1}) \stackrel{def}{=} s_i.o = s_i.i_1 \times s_i.i_2 \wedge s_{i+1}.o = s_{i+1}.i_1 \times s_{i+1}.i_2 \,.$$

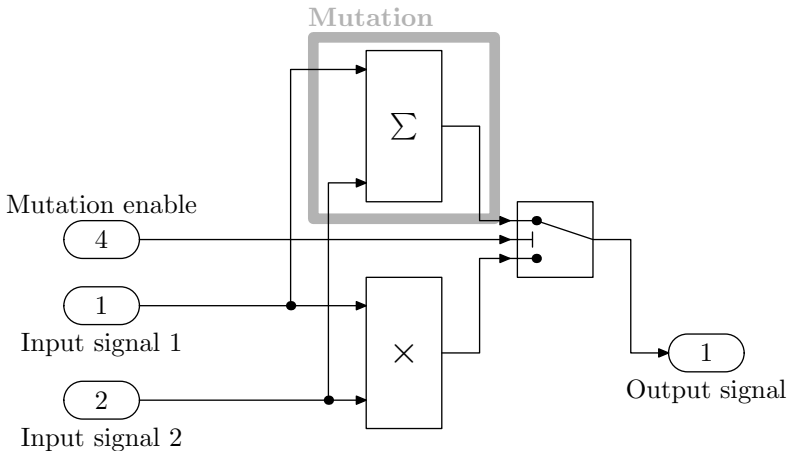
A possible syntactic mutation is to replace the multiplication ($\times$) with an addition:

$$R'(s_i, s_{i+1}) \stackrel{def}{=} s_i.o = s_i.i_1 + s_i.i_2 \wedge s_{i+1}.o = s_{i+1}.i_1 + s_{i+1}.i_2 \,.$$

This mutation can be implemented in the diagram using an *enable* signal, allowing us to switch the mutation on and off (see Figure 4b).



(a) A simple Simulink diagram



(b) A Simulink diagram with a mutation

Fig. 4: A simple Simulink program and its mutation

In our formalism, such a modification can be modeled 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., occur repeatedly). The former case can be simply modeled by permanently altering the transition relation R, and applying the resulting relation R' in each step:

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

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. Accordingly, a typical execution satisfies the following constraint:

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

Transient or intermittent faults can be modeled by referring to a global timer. Let R' be a transition function with a permanently enabled mutation. Furthermore, given a state s_i, let $s_i.t$ denote a signal tracking progression of time during

the execution. A transition function R'' with an intermittent fault occurring every c execution steps can be modeled as

$$R''(s_i, s_{i+1}) \stackrel{def}{=} \begin{cases} R'(s_i, s_{i+1}) & \text{if } (s_i.t = 0 \bmod c) \\ R(s_i, s_{i+1}) & \text{if } (s_i.t \neq 0 \bmod c) \end{cases} .$$

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

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

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 behavior. 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 or k-induction. A brief outline of these techniques is provided in Section 5.1.

3.6 Generating Test Cases

One way of generating test cases that detect a mutation is to find a satisfying assignment to Formula (1). Such a satisfying assignment provides the inputs that yield a different output sequence during the first k steps and, provided the observable behaviors of the two models M and M' are not fully equivalent, such a solution must exist for some k.

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 (1) for each pair of models M and M'. In practice, this approach is very wasteful, since modern satisfiability checkers such as MiniSAT [27] 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 Figure 4b 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:

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

We use M_μ to denote the model with the transition relation R_μ.

Given the resulting transition relation R_μ (as defined in (2)) we can construct an instance of Formula (1). A fault in the modified model M_μ can be triggered by adding a constraint of the form

$$F^j := f^j \wedge \bigwedge_{0<i\leq n, i\neq j} \neg f^i \ . \tag{3}$$

Example 2. Figure 5 shows a mutation and a fault injected into the Simulink diagram in Figure 4a. The first diagram shows the implementation of a signal-stuck-at-0 fault. The diagram below combines this fault and the mutation in Figure 4b into one model. The model provides two flags allowing us to trigger the mutations.

The decision procedure of CBMC, the model checker which our test case generation tool COVER is based on, performs bit-level accurate reasoning by transforming the instance of Formula (1) 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 [27]. The decision process of MINISAT is incremental, i.e., it allows

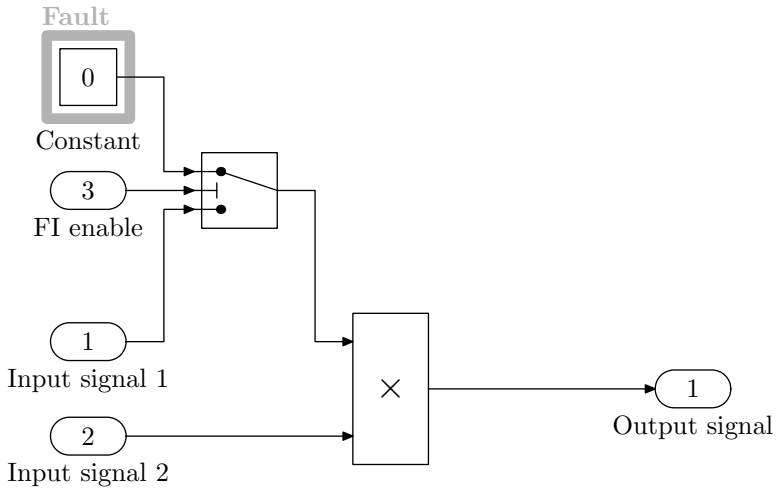
- to add additional clauses, and
- to add or remove a constraint F^j of the form described in (3)

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

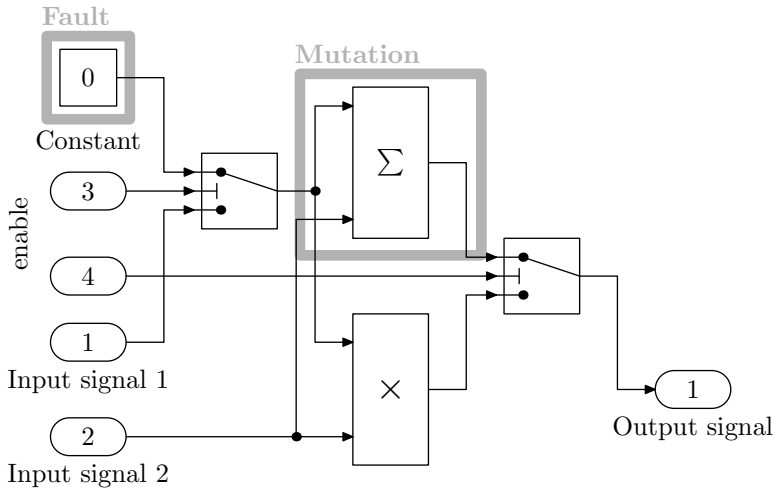
Let EQ_k denote the CNF of the instance of Equation (1) derived from the models M and M_μ. We can generate a test suite covering all n mutations in question by iteratively computing satisfiable assignments to

$$EQ_k \wedge F^i, \quad i \in \{1, \ldots, n\} \ . \tag{4}$$

[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.



(a) Fault Injection: A signal-stuck-at-0 fault



(b) The fault and the mutation from Figure 4b combined in one model

Fig. 5: Mutations and Faults injected into a simple Simulink model

We say that a test case t *independently* covers a mutation if it corresponds to a satisfying assignment of (4) for that mutation i. If each of the instances of Formula (4) has a solution, we obtain n (not necessarily different) test cases that cover all mutations injected into the model M_μ.

4 Generating Test-Cases for Many Mutations

In this section, we propose an optimisation of the mutation-based test-case generation approach discussed in Section 3.

4.1 Finding an Efficient and Sufficient Test-Suite

The technique described in Section 3.6 uses a SAT-solver to extract n test cases from Formula (4), each of which corresponds to one mutation. If the number of mutations (n) is large, this may lead to an equally large number of test cases. These test-cases are computationally expensive to generate and time-consuming to execute. Therefore, it is desirable to minimize the size of the test-suite.

Example 3. Consider the Simulink diagrams in Figures 4a, 4b, and 5. Assume that we use the model in Figure 5b to generate a test case. Consider the test case $s.i_1 = 1.2$ and $s.i_2 = 23.4$. The following table lists the observable outputs of the model for all possible combinations of enable flags:

Input signal 1	**Input signal 2**	**Fault**	**Mutation**	**Output**
1.2	23.4	off	off	28.08
1.2	23.4	on	off	0.0
1.2	23.4	off	on	24.6
1.2	23.4	on	on	23.4

This test case is sufficient to detect the syntactic mutation, the signal-stuck-at-0 fault, as well as the combination of these modifications.

The observation in Example 3 suggests that it is not strictly necessary to generate a separate test case for each single modification. In the setting presented in Example 3, the test vector $t = \{i_1 \mapsto 1.2, i_2 \mapsto 23.4\}$ can be obtained from the model in which *both* modifications are enabled. This can be achieved by generalizing the constraint F^j (see (3)) accordingly, i.e., for a set T of indices corresponding to mutations or faults

$$F^T := \bigwedge_{0<i\leq n} \big((i \in T) \Rightarrow f^i\big) \wedge \big((i \notin T) \Rightarrow \neg f^i\big) \ . \tag{5}$$

Notably, t is sufficient to cover the fault and the mutation independently.

It follows that it is possible that a test-case t derived from a model M_μ with a combination of several mutations $\{\nu_1, \nu_2, \ldots\}$ detects the independent mutations, too. This can be efficiently checked by evaluating the behavior of the mutated models $M_{\nu_1}, M_{\nu_2}, \ldots$ for the input defined by the test-vector t. The execution of a given test-case on a model is very efficient compared to the model checking-based computation of a new test-vector for M_{ν_i}.

Example 4. We continue working in the setting of Example 4. Consider the test case $s.i_1 = 0.0$ and $s.i_2 = 23.4$. We can compute the outcome for all different models for this input by simply executing the test cases:

Input signal 1	**Input signal 2**	**Fault**	**Mutation**	**Output**
0.0	23.4	off	off	0.0
0.0	23.4	on	off	0.0
0.0	23.4	off	on	23.4
0.0	23.4	on	on	23.4

This test case is sufficient to detect the combination of the syntactic mutation and the injected signal-stuck-at-0 fault, as well as the single syntactic mutation. However, it fails to detect the signal-stuck-at-0 fault.

Example 4 shows that it is not always the case that a test-case that detects a combination of mutations and faults also covers each mutation independently. While we have found a test vector $t = \{i_1 \mapsto 0.0, i_2 \mapsto 23.4\}$ which "kills" the mutation, the same test vector fails to cover the fault. We are forced to generate an additional test case for the injected fault, which can be achieved by deactivating the enable flag for the fault and starting another incremental run of the SAT solver.

We propose to analyze the model systematically, starting with a set of mutations T. The corresponding algorithm is outlined in Figure 6. We generate a test case which covers this set of mutations T and check whether it independently covers the single elements of T. The advantage of the algorithm is that it is possible to prune entire subsets of mutations if C in step ③ is non-empty. If this is not the case, we have to split T recursively. In the worst case, we still require n test-cases to cover all n mutations.

Input: A model and a set of mutations T
Output: A test-suite S covering the mutations in T.

① If $T = \emptyset$, terminate.
② Compute a test case t satisfying $EQ_k \wedge F^T$ (as defined in (5)).
($C = \emptyset$ if there is no such t).
③ Let $C \subseteq T$ be the set of mutations independently covered by t:
❶ If $C \neq \emptyset$, let $T := T \setminus C$. Add t to the test suite and proceed to ①.
❷ If $C = \emptyset$, partition T into T_1 and T_2 s.t. $T = T_1 \cup T_2$, and $T_1 \cap T_2 = \emptyset$.
Call the algorithm recursively with $T := T_1$ and $T := T_2$, respectively.

Fig. 6: An algorithm for systematic test-case generation.

The success or failure of this approach depends inherently on the structure of the mutations and the model M. In the following, let ν_1 and ν_2 be two mutations for M. Furthermore, let $s_0, \ldots, s_n$ be an execution trace of M, and $s'_0, \ldots, s'_n$ be an execution trace of M with both mutations enabled.

Assume that ν_1 and ν_2 affect different output signals, i.e., given a fixed input sequence $s_0.i, \ldots s_n.i$, the set of signals changed by activating the mutation ν_1 is disjoint from the set of signals changed by enabling the mutation ν_2. Then, we can increase the chance of finding a test case that covers both mutations independently by trying to maximize the difference between $s_0.o, \ldots s_n.o$ and $s'_0.o, \ldots s'_n.o$.

Unfortunately, such an independence cannot be assumed in general, since there may be a mutual influence between mutations. In particular, two mutations may cancel each other out. Checking whether two mutations are independent in the sense explained above is computationally as expensive as model checking and therefore not a feasible strategy.

5 Detecting Non-Observability of Mutations

In traditional mutation-based testing, the difficulty to identify mutations without observable effect on the system outputs is known to be one of the main obstacles. Assume that one instance $EQ_k \wedge F^c$ of (4) (with $i = c$) is unsatisfiable. This indicates that the injected fault corresponding to f^c (see Formula (2)) 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 behavior. We say that the model *tolerates* the fault. The mutant is an *equivalent mutant*.

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 mutation is not *strong* enough to have any impact on the observable behavior of the model and the model checking tool provides a *proof* for the equivalence of the mutated and original model. This concept is explained in the following subsection.

5.1 Model Checking, Induction, and Invariants

BMC is only capable of providing a guarantee that a property P is not violated within at most k execution steps. In this section, we briefly discuss two techniques to lift this restriction: k-induction [28] and finding invariants by means of fixed-point detection [1].

k-Induction. This technique generalizes the standard induction principle, and has been used before for test-vector generation for Simulink models [2]. The base case is established by means of BMC. The following equation holds if and only if the property P is not violated in the first k execution steps:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k} P(s_i) \tag{6}$$

If the base case (6) holds, the technique proceeds to show by induction that P holds for any arbitrary $k \in \mathbb{N}$:

$$\left(\bigwedge_{i=0}^{k} R(s_i, s_{i+1}) \wedge \bigwedge_{j=0}^{k} P(s_j) \right) \Rightarrow P(s_{k+1}) \tag{7}$$

Formula (6) in combination with (7) implies that the sequence of states can be extended to a path of arbitrary length without ever violating P. Thus, if we can find a k for which the conjunction of (6) and (7) holds, then the model is safe.

To check whether this conjunction holds for a given k, we rely on efficient decision procedures (SAT solvers such as [27], in particular). Let $G = (6) \wedge (7)$. If G holds, modern decision procedures are able to generate a *proof*. In our setting, a proof is a directed acyclic graph (V, E, ℓ), where V is a set of vertices, E is a set of edges, and ℓ is a labeling function. Each initial vertex v has in-degree 0 and $\ell(v)$ is an axiom or a sub-conjunct of G. Each internal vertex has in-degree m, $m \geq 1$. The label of each inner node w is derived from the labels of its predecessors $\{v_1, \ldots, v_m\}$ by means of a deduction rule $\ell(v_1), \ldots, \ell(v_m) \vdash \ell(w)$. The final vertex u has out-degree 0 and $\ell(u)$ is the conclusion of the proof. SAT solvers typically generate proofs that $\neg G$ is unsatisfiable, that is, the conclusion is F.

Fixed-Points and Invariants. For finite-state systems, iterating the transition function until no new states are found is a viable verification technique, known as fixed-point detection. This technique relies on an efficient symbolic representation of sets of states such as binary decision diagrams [29]. The following recursive equations are iterated until $\mathcal{S}_i = \mathcal{S}_{i+1}$:

$$\mathcal{S}_0 = \{s_0 | I(s_0)\}, \quad \mathcal{S}_{i+1} = \mathcal{S}_i \cup \{s_{i+1} | s_i \in \mathcal{S}_i \wedge R(s_i, s_{i+1})\} \tag{8}$$

Let J be a symbolic representation of the final $\mathcal{S}_i$ in this sequence. Then J is an inductive invariant, i.e., it holds that

$$I(s_0) \Rightarrow J(s_0)\,, \quad \text{and} \quad J(s_i) \wedge R(s_i, s_{i+1}) \Rightarrow J(s_{i+1})\,. \tag{9}$$

A popular technique to find invariants is the over-approximation of the set of safe states by means of Craig interpolation [30].

5.2 Reusing Proofs and Invariants for Proving Unobservability

Proof Analysis. The techniques presented in this section are based on work recently presented by Purandare et al. [31]. Consider a system M and a mutated system M_o in which all the mutations are introduced, but disabled using $F^{\emptyset}$, i.e. $\forall i. \neg f^i$. Thus, M and M_o are equivalent, i.e. $s_i.o = s'_i.o$ at all times. The equivalence checking Formula (1) is unsatisfiable for all k. As stated in Section 5.1, k-induction is one possibility to prove equivalence of M and M_o. Let T be a set

of indices corresponding to all possible mutations (cf. Section 4.1). The formulae that k-induction checks to establish equivalence of M and M_o are as follows:

$$\underbrace{I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})}_{\text{original model}} \wedge \underbrace{I'(s'_0) \wedge \bigwedge_{i=0}^{k-1} R'(s'_i, s'_{i+1})}_{\text{mutated model}} \wedge \underbrace{\bigwedge_{j \in T} \neg f^j}_{\text{disable T}} \wedge \underbrace{\bigwedge_{i=0}^{k} s_i.o = s'_i.o}_{\text{equality of output}} \tag{10}$$

$$\left(\bigwedge_{i=0}^{k} (R(s_i, s_{i+1}) \wedge R'(s'_i, s'_{i+1})) \wedge \bigwedge_{j \in T} \neg f^j \wedge \bigwedge_{j=0}^{k} s_j.o = s'_j.o \right) \Rightarrow s_{k+1}.o = s'_{k+1}.o \tag{11}$$

The following theorem states that those mutations that do not appear in the final proof of $G = (10) \wedge (11)$ checked during k-induction are non-observable [31].

Theorem 1. *A mutation corresponding to an enabling variable f^j that does not appear in the final proof of G is unobservable.*

Proof. A variable absent in the proof does not influence the conclusion of the proof. Thus, enabling a flag f^j absent in the proof does not invalidate the proof of k-inductivity and hence, does not break equivalence.

Thus, mutations absent in the proof do not need to be checked as these are unobservable.

Inductive Invariant. An inductive invariant represents a set of safe states with respect to a certain property of the model. Consider the composition of the two models M and M_o (where $\forall i \,.\, (s_i.i = s'_i.i)$) and let $\forall i \,.\, (s_i.o = s'_i.o)$ be the property in question.

Theorem 2. *Let $J(s, s')$ represent an invariant over the state space of the two models that warrants the equivalence of M and M_o, i.e., that $\forall i \,.J(s_i, s'_i) \Rightarrow (s_i.o = s'_i.o)$ holds. Furthermore, let R' be the transition function of M_o, only that the j^{th} mutation is enabled ($f^j = \mathsf{T}$). If*

$$I(s_0) \wedge I'(s'_0) \Rightarrow J(s_0, s'_0) \;\; \textit{and}$$
$$J(s_i, s'_i) \wedge R(s_i, s_{i+1}) \wedge R'(s'_i, s'_{i+1}) \Rightarrow J(s_{i+1}, s'_{i+1})$$

holds, then the j^{th} mutation is unobservable.

Proof. The inductive invariant J is also an inductive invariant of the resulting mutated system and therefore establishes equivalence.

6 Conclusion

We have defined a methodology for automated test case generation for Simulink models. By formulating test coverage and goals in terms of fault models, we

achieve a flexible and general framework that subsumes standard coverage criteria and is directly related to functional and non-functional requirements specifications. The use of equivalence checking and bounded model checking makes it possible to explore the behavior of models with high precision, taking intricate details such as the actual floating-point semantics of execution platforms into account. We implemented this approach in our test-case generation tool COVER, which is based on the model checker CBMC. The evaluation of COVER on industrial case studies developed in the European projects MOGENTES and CESAR is currently in progress.

In order to handle the size of real-world Simulink models, we have introduced two main concepts to keep the complexity of test case generation manageable: a strategy to compute small test suites by maximizing the number of mutations that are covered by each test case, and techniques to efficiently detect unobservability of mutations. An experimental evaluation of both techniques is planned as future work.

References

1. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
2. Gadkari, A., Yeolekar, A., Suresh, J., Ramesh, S., Mohalik, S., Shashidar, K.C.: AutoMOTGen: Automatic model oriented test generator for embedded control systems. In Gupta, A., Malik, S., eds.: Computer Aided Verification (CAV). Volume 5123/2008 of LNCS., Springer (2008) 204–208
3. Kroening, D., Clarke, E.M., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Design Automation Conference (DAC), ACM (2003) 368–371
4. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic test case generation for dynamic analysis and measurement. In: Computer Aided Verification (CAV). Volume 5123 of LNCS., Springer (2008) 209–213
5. Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Automatic test generation for coverage analysis using CBMC. In: Computer Aided Systems Theory (EUROCAST). Volume 5717 of LNCS., Springer (2009) 287–294
6. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Verification, Model Checking and Abstract Interpretation (VMCAI). Volume 5403 of LNCS., Springer (2009) 151–166
7. Ball, T.: A theory of predicate-complete test coverage and generation. In: Formal Methods for Components and Objects (FMCO). Volume 3657 of LNCS., Springer (2005) 1–22
8. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: International Conference on Software Engineering (ICSE). (2004) 326–335
9. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering (TSE) (2010)
10. Kupferman, O., Li, W., Seshia, S.A.: A theory of mutations with applications to vacuity, coverage, and fault tolerance. In: Formal Methods in Computer-Aided Design (FMCAD), IEEE (2008) 1–9
11. Ruthruff, J.R., Burnett, M.M., Rothermel, G.: Interactive fault localization techniques in a spreadsheet environment. IEEE Transactions on Software Engineering (TSE) **32** (2006) 213–239

12. Schuler, D., Dallmeier, V., Zeller, A.: Efficient mutation testing by checking invariant violations. In: International Symposium on Software Testing and Analysis (ISSTA), ACM (2009) 69–80
13. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Formal Engineering (ICFEM). Volume 4260 of LNCS., Springer (2006) 606–620
14. Fehnker, A., Krogh, B.H.: Hybrid system verification is not a sinecure: The electronic throttle control case study. In: Automated Technology for Verification and Analysis (ATVA). Volume 3299 of LNCS., Springer (2004) 263–277
15. Joshi, A., Heimdahl, M.P.E.: Model-based safety analysis of Simulink models using SCADE design verifier. In: Computer Safety, Reliability, and Security (SAFECOMP). Volume 3688 of LNCS., Springer (2005) 122–135
16. Ryabtsev, M., Strichman, O.: Translation validation: From simulink to c. In: Computer Aided Verification (CAV). Volume 5643 of LNCS., Springer (2009) 696–701
17. The Mathworks: Simulink design verifier user's guide. `http://www.mathworks.com/access/helpdesk/help/toolbox/sldv/` (2009) version 1.5.
18. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: Formal Methods in Computer-Aided Design (FMCAD), IEEE (2009) 69–76
19. Kuehlmann, A., van Eijk, C.A.J.: Combinational and sequential equivalence checking. In: Logic Synthesis and Verification. Kluwer International Series in Engineering and Computer Science Series. Kluwer, Norwell, MA, USA (2002) 343–372
20. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Computer Aided Verification (CAV). Volume 1254 of LNCS. Springer (1997) 72–83
21. Kroening, D., Clarke, E.: Checking consistency of C and Verilog using predicate abstraction and induction. In: IEEE/ACM International conference on Computer-aided design, IEEE (2004) 66–72
22. Victor A., C.: Interpretation of IEEE-854 floating-point standard and definition in the HOL system. Technical report, NASA Langley (1995)
23. Harrison, J.: Formal verification of square root algorithms. Formal Methods in System Design (FMSD) **22** (2003) 143–153
24. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation (PLDI), ACM (2003) 196–207
25. Min, A.: Relational abstract domains for the detection of floating-point run-time errors. In: European Symposium on Programming (ESOP). Volume 2986 of LNCS., Springer (2004) 3–17
26. Kroening, D., Strichman, O.: Decision Procedures. Springer (2008)
27. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing (SAT). Volume 2919 of LNCS., Springer (2004) 502–518
28. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Formal Methods in Computer-Aided Design (FMCAD). Volume 1954 of LNCS., Springer (2000) 108–125
29. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **35** (1986) 677–691
30. McMillan, K.L.: Interpolation and SAT-based model checking. In: Computer Aided Verification (CAV). Volume 2725 of LNCS., Springer (2003) 1–13
31. Chockler, H., Kroening, D., Purandare, M.: Coverage in interpolation-based model checking. In: Design Automation Conference (DAC), ACM (2010)