Test-Case Generation for Embedded Simulink via Formal Concept Analysis*

Nannan He
Oxford University

Philipp Rümmer
Uppsala University

Daniel Kroening
Oxford University

ABSTRACT

Mutation testing suffers from the high computational cost of automated test-vector generation, due to the large number of mutants that can be derived from programs and the cost of generating test-cases in a white-box manner. We propose a novel algorithm for mutation-based test-case generation for Simulink models that combines white-box testing with formal concept analysis. By exploiting similarity measures on mutants, we are able to effectively generate small sets of short test-cases that achieve high coverage on a collection of Simulink models from the automotive domain. Experiments show that our algorithm performs significantly better than random testing or simpler mutation-testing approaches.

Keywords

mutation-based test-case generation; embedded software; Simulink; change impact analysis; concept lattice;

1. INTRODUCTION

Mutation coverage is a method to quantify the quality of a test suite for a design artifact, such as a circuit or a software program. The key idea is to define a set of pre-defined *mutation operators*, which are small, systematic modifications of the representation of the design artifact [5, 11]. The intent is to capture typical design errors, such as the choice of an inappropriate arithmetic operator. The design that results from the application of a mutation operator is called the *mutant*. A test suite is considered good if it contains tests that are able to distinguish a large number of these mutants from the original design. Given a sufficiently rich set of mutation operators, mutation coverage subsumes many other popular notions of coverage, such as location and modified condition/decision coverage (MC/DC) for software and stuck-at faults for hardware [14].

Mutation testing is a computationally expensive technique, for a number of reasons. An obvious cause of computational cost is the sheer number of mutants that can be considered. Mutations are small local changes to parts of the model representation. Given a set of m mutation operators and a model with n components (e.g., lines in a program or gates in a netlist), a maximum of $m{\cdot}n$ mutants can be constructed. If simultaneous mutation of multiple locations of a model is considered, the number of mutants even grows exponentially in the number of applied mutations. Another cause of computational cost are *equivalent mutants*, which are mutated designs that are actually indistinguishable from the original design [5]. It is often too difficult or not feasible to algorithmically decide whether a mutant is equivalent, so that much time can be wasted searching for a covering test.

This paper focuses on automated test case generation for a specific class of designs: we specialize on dataflow models given in the Simulink design language. The Simulink framework is the predominant modeling formalism for embedded control software in the automotive industry, and is also widely deployed in other safety-critical domains, such as for avionic applications. In previous work [2], we have described a tool chain for generating test vectors for Simulink models, given individual mutants of the model. 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. The method presented in [2] relies on a bounded model checking (BMC) engine for software [3]. The model and the mutant are encoded into a decision problem for a SAT-based decision procedure for a given execution depth. The formula is satisfiable iff the mutant can be discovered by a trace of the given depth. The application of model checking or constraint-based techniques to generate high-coverage test suites has become commonplace (see, for instance, [6] for an application in the automotive domain).

In our experiments we observe that BMC is effective at discovering short tests for a given mutant, but that the computational cost of the model checking engine is high. This renders the naïve way of applying the model checker to each individual mutant ineffective for any non-trivial number of mutants.

This paper proposes a new algorithmic technique for automated mutation testing for Simulink. The key to an effective use of the expensive model checking procedure is to exploit the structure of the Simulink diagram. Simulink diagrams can be seen as graphs in which the nodes represent computational blocks and the edges describe the flow of data. For this kind of program representation with explicit dataflow it is easy to perform a conservative *change impact analysis*,

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

DAC 2011, June 5-10, 2011, San Diego, California, USA.

i.e., we can compute a (conservative) subset of nodes that can possibly observe any given mutation at low cost. Subsequently, this information can be exploited to analyze a large set of mutants simultaneously with a single call to the model checking engine: on our benchmark set, we have observed a case of 5 (short) tests that are able to detect 108 mutants.

The main contributions of this paper are

1. a generic mutation testing framework for Simulink programs that is flexible w.r.t. a number of design decisions, including mutant selection strategies and coverage checks;

2. a theoretical and empirical comparison of various instances of the framework using an industrial case study;

3. heuristics derived from formal concept analysis that optimize the use of the framework.

Related Work.

A number of papers report applications of CBMC or similar techniques for generating high-coverage test suites [8, 1, 16]. 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 and Blast. We only consider mutant models with single mutations, whereas other authors also consider combinations of mutations [13]. Ruthruff et al. [15] propose to use mutations to prioritize test-cases to increase a test suite's rate of fault detection. We propose ordering mutants instead of test-cases, to both reduce the TCG cost and minimize the size of test suite.

To the best of our knowledge, there is no previous work on the use of formal concept analysis to support mutation testing. An application of other clustering methods, such as K-means clustering, is described in [9]. Our method has similarities with structural fault collapsing [10], where one representative fault is chosen from a set of faults that are structurally proximal and prove equivalent.

2. BACKGROUND

2.1 Matlab Simulink

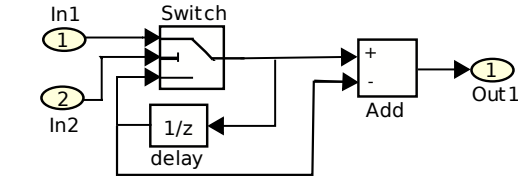
Matlab Simulink is a graphical dataflow language that is commonly used in an industrial context for modeling or implementing control applications. Simulink models consist of a set of *blocks* that are connected by *signals* specifying the flow of data. Blocks are taken from pre-defined block libraries (covering generic functions such as addition or logical operators, but also domains like fuzzy logic or network communication) and receive a specific number of input signals from which output signals are computed. Stateful systems are modeled with the help of feedback loops. Models can be structured hierarchically with the help of *subsystems,* and can be simulated, analyzed, or compiled to code using the Matlab tool-suite and third-party products.

For the purposes of this paper, we only consider time-discrete Simulink models, which means that signals represent (potentially infinite) streams of values governed by a global clock.[1] The semantics of blocks is synchronous in the sense that every block is evaluated and performs exactly one computation step per time unit. As a whole, a Simulink program receives a number of (potentially infinite) streams of input values (specified using *inports* in the Simulink model) and generates a number of output streams (described using *outports*). Fig. 1 gives an example of a Simulink model.

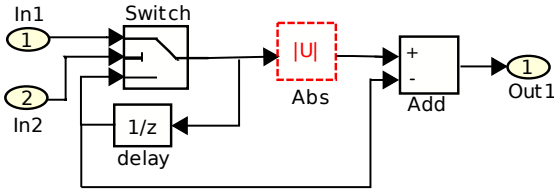
[1]We believe that our results can be generalized to time-continuous and event-driven diagrams, with appropriate changes to the techniques described in Sect. 4.

Acronym	Description
RC	Replace Constant x with $x+1$, $x-1$, or 0
ABS	Insert absolute value operator
UOI	Insert negation ($-$, $\neg$) operator
INC	Add constant value to a signal
RR	Swap relational operators $<, \leq, >, \geq, =$
RL	Swap Boolean operators $\wedge$ and $\vee$

Table 1: Mutations applied to Simulink models



(a) Original Simulink model



(b) Mutated Simulink model

Figure 1: Example of a mutated Simulink model

2.2 Mutation-based test-case generation

In this paper, we consider test-case generation (TCG) strategies for Simulink models built on top of the mutation-based TCG approach defined in [2], which uses bounded model checking techniques to systematically construct test-cases. *Mutation-based TCG* proceeds by injecting syntactic mutations (in this context sometimes also called *faults*) into a given Simulink model S, generating from S a set M of *mutants*. The types of mutation operators considered in this paper are given in Table 1. The goal of TCG is to find a set of test-cases (finite sequences of inputs for the models S) that *kill* each of the mutants $S' \in M$, which means that the test-case makes the mutant S' produce outputs that differ from those of the original model S. The main hypothesis underlying mutation testing is that such test-cases, which are able to detect simple bugs like the injected syntactic mutations, are also useful for finding real, potentially more complicated defects (this is called the *coupling effect* [5]).

We show an example of a mutated Simulink model in Fig. 1b, illustrating the "Absolute Value" (ABS) mutation operator. Some of the mutations (like ABS) are applied by inserting additional blocks into a diagram, while other mutations replace existing blocks with new blocks.

In the style of bounded model checking [4], both the original model S and each of its mutants $S' \in M$ can be modeled using transition relations R and R' and formulae I, I' defining the initial states. As in sequential equivalence checking [12], observational equivalence of S and S' during the

first k computation steps can then be expressed using the following formula:

$$\underbrace{\bigwedge_{i=0}^{k} s_i.i = s'_i.i}_{\text{equality of all inputs}} \quad \wedge \underbrace{I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge}_{\text{original model } S}$$

$$\underbrace{I'(s'_0) \wedge \bigwedge_{i=0}^{k-1} R'(s'_i, s'_{i+1}) \wedge}_{\text{mutant } S'} \quad \underbrace{\bigvee_{i=0}^{k} s_i.o \neq s'_i.o}_{\text{inequality of an output}} \qquad (1)$$

Any satisfying assignment for this formula represents two executions of S and S' that yield a different output sequence; the projection of the assignment to the inputs corresponds to a test-case. As most Simulink models operate on scalar datatypes such as machine integers or floating-point arithmetic, and therefore have a finite state space, satisfying assignments can be constructed using SAT/SMT-based techniques.

3. SIMULINK TCG STRATEGIES

3.1 The TCG procedure

The inherent difficulty of mutation-based TCG is the vast number of mutants that can be generated from real-world programs: this number grows polynomially in the size of the program and the number of considered mutation operators, and exponentially in case multiple mutations are applied simultaneously. The use of efficient subsumption criteria and heuristics is therefore indispensable in any TCG approach. We address this need systematically by defining a TCG framework that can be instantiated using various ordering and subsumption strategies:

Algorithm 1: The general TCG procedure

Input: A model S and a set M of mutants
Output: A test-suite T for the model S

$T \leftarrow \emptyset$;
$M_r \leftarrow M$;
while $M_r \neq \emptyset$ **do**
  pick mutant $S' \in M_r$; // (*)
  generate test-case t that kills S'; // (**)
  $T \leftarrow T \cup \{t\}$;
  $M_r \leftarrow M_r \setminus \{S'\}$;
  remove further mutants from M_r; // (***)
end
return T;

The procedure is parametric in three respects:

(*) Mutants can be selected using different strategies; this is discussed in Sect. 3.2 and 4.

(**) We use the approach described in Sect. 2.2 to generate test-cases for individual mutants with the help of BMC. More refined methods might consider multiple mutants at the same time, but are beyond the scope of this paper.

(***) With each new test-case, killed mutants can be removed from the set M_r of remaining mutants. The most accurate method is to execute each remaining mutant for the new test-case; more efficient (heuristic) approaches are discussed in Sect. 4.2.4.

Note that (**) might fail to generate a test-case killing S', leading to an unchanged suite T in this iteration of the procedure. Since S' will be removed from M_r, termination of the procedure is guaranteed also in this case.

The rest of the paper will discuss different choices of (*), (**), (***), with the following objectives:

O1) Minimize the size of test-suites T.

O2) Maximize coverage, i.e., find test-suites T that kill as many mutants in M as possible.

O3) Minimize the runtime of the procedure; this primarily means that the main loop of Alg. 1 should be executed as few times as possible.

3.2 Simple mutant selection strategies

Mutant selection (*) aims at selecting those mutants first that yield good test-cases (which are test-cases that kill a large number of mutants), but at the same time should prefer mutants that are not too difficult to kill; otherwise, TCG (**) might consume a lot of time. We propose and evaluate the following "naïve" strategies to this end:

3.2.1 Random mutant selection

As baseline approach, mutants can be picked randomly from M_r in (*).

3.2.2 Mutations close to observation points (MCTO)

Naturally, mutations close to the outports of a Simulink model are easy to kill by test-cases, because the effects of the mutation are likely to be carried through to the outports. Given a Simulink model S with outports $O_1, \ldots, O_n$, and a location l in S, we can compute the lengths $d_1(l), \ldots, d_n(l)$ of shortest paths from l to $O_1, \ldots, O_n$, respectively. This induces the *observation distance* order $\preceq_{\text{od}}$ on model locations

$$l \preceq_{\text{od}} l' \quad \equiv \quad \{\!\{d_1(l), \ldots, d_n(l)\}\!\} \preceq_{\text{ms}} \{\!\{d_1(l'), \ldots, d_n(l')\}\!\}$$

where the order $\preceq_{\text{ms}}$ on multisets of integers is defined by

$$\{\!\{\alpha_1, \ldots, \alpha_k\}\!\} \preceq_{\text{ms}} \{\!\{\beta_1, \ldots, \beta_m\}\!\} \equiv \langle \alpha_1, \ldots, \alpha_k \rangle \preceq_{\text{lex}} \langle \beta_1, \ldots, \beta_m \rangle$$

with $\alpha_1 \leq \cdots \leq \alpha_k$, $\beta_1 \leq \cdots \leq \beta_m$, and $\preceq_{\text{lex}}$ is the lexicographic order of tuples of integers.

The MCTO strategy for (*) will randomly pick one of the mutants in M_r for which the location of the injected mutation is minimal in the $\preceq_{\text{od}}$ order.

3.2.3 Mutations far from observation points (MFFO)

Inverting MCTO, the MFFO strategy will randomly pick mutants for which the location of the applied mutation is maximal in the observation distance order $\preceq_{\text{od}}$. This means that mutations far away from outports are preferred. Such mutants are normally harder to kill than mutants obtained through mutations close to the observation points, but individual test-cases resulting from MFFO might be more useful than those from MCTO: since MFFO test-cases guarantee that the modified signal is passed through a large part of the model under test, it is more likely that also other mutations become observable. Similarly, following the coupling

hypothesis, we can argue that MFFO mutations correspond to deeper or more intricate bugs in a design.

4. APPLYING CONCEPT ANALYSIS

4.1 Overview of formal concept analysis

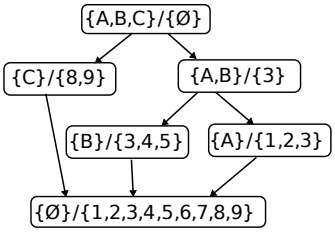
Formal concept analysis (FCA) [7] is a means of categorizing objects based on attributes, making it possible to systematically identify similarities and differences by constructing a hierarchy of object groups. In the context of TCG, FCA can be used to cluster and order mutants, and thus drive the steps `(*)` and `(***)` of Alg. 1. On the most abstract level, FCA is applied to contexts (O, A, I) consisting of a set O of *objects* (in our case, mutants), a set A of *attributes*, and a binary relation $I \subseteq O \times A$ between objects and attributes. A *concept* is a pair of sets (X, Y) satisfying the following equations:

$$X = \{o \in O \mid \forall a \in Y : (o, a) \in I\} \tag{2}$$
$$Y = \{a \in A \mid \forall o \in X : (o, a) \in I\} \ . \tag{3}$$

X is called the *extent* of the concept, while Y is called the *intent*.

Object	Attributes
A	1,2,3
B	4,5,3
C	8,9

(a) Context



(b) Concept Lattice

Figure 2: Example of a concept lattice

Concepts can be compared using the partial order $\leq$ defined by

$$(X, Y) \leq (X', Y') \quad \equiv \quad X \subseteq X'$$

which gives the set of all concepts over a context (O, A, I) the (uniquely defined) structure of a complete lattice. Note that the inclusions $X \subseteq X'$ and $Y' \subseteq Y$ are equivalent.

Fig. 2 gives an example context and the resulting concept lattice. The top concept represents the set of all objects (and the set of attributes common to all objects), and the bottom concept the set of all attributes (and the set of objects that possess all attributes).

4.2 Clustering mutants using concept lattices

To analyze interdependencies in a set M of mutants of a Simulink model S, we consider the concept lattices generated by contexts (M, A, I). The goal is to group together those mutants that behave similarly with respect to testing; by "similar," we mean that the mutants are likely to be killed by the same test-cases. This kind of clustering can be achieved by selecting a set A of attributes representing testing-relevant properties of mutants, e.g., the mutation operator or the mutated location in S.

4.2.1 Perfect clustering

To illustrate this approach, we describe (theoretically) the perfect concept lattice for clustering mutants. Because this perfect lattice is too expensive or impossible to compute, it will later be approximated using coarser sets of attributes.

For perfect clustering, we define the attributes A to be the (usually infinite) set of input sequences that the model S can receive:[2]

$$A_{\text{Inp}} \ = \ (IP \to \mathcal{D})^*$$

Here, IP represents the set of all inports of S, and $\mathcal{D}$ the data domain from which inputs are taken. The relationship between mutants and attributes is defined by:

$$I_{\text{Inp}} \ = \ \{(S', a) \in M \times A_{\text{Inp}} \mid a \text{ kills } S'\}$$

The concepts generated by the context $(M, A_{\text{Inp}}, I_{\text{Inp}})$ group together exactly those mutants that are killed by the same model inputs, and are thus useful for selecting inputs representing good test-cases. Clearly, if the top concept of the resulting concept lattice happens to have a non-empty intent, any input taken from the intent is an "optimal" test-case that kills *all* mutants in M. If the intent of the top concept is empty, no single test-case suffices, but we can still extract test-cases killing many mutants simultaneously from the maximal concepts with non-empty intent.

4.2.2 Approximative clustering

The concept lattice described in the previous paragraphs can obviously not be computed in practice, but it can be approximated in order to achieve a similar classification of mutants. The resulting lattices can be traversed in various ways in order to implement steps `(*)` and `(***)` of Alg. 1.

In the scope of this paper, we concentrate on concepts classifying mutations according to their *cone of influence* in the Simulink model S (the set of blocks that values originating from the mutation can reach). This follows the hypothesis that mutations applied to similar locations in S are likely to be killed by the same test-cases:

$$A_{\text{Blk}} \ = \ \{a \mid a \text{ is a block in } S\} \tag{4}$$
$$I_{\text{Blk}} \ = \ \left\{(S', a) \;\middle|\; \begin{array}{l} a \text{ is reachable from the} \\ \text{mutated location in } S' \end{array}\right\} \tag{5}$$

Concepts (X, Y) resulting from the context $(M, A_{\text{Blk}}, I_{\text{Blk}})$ represent mutants X whose mutations share the influenced blocks Y. Concepts near the bottom element of the lattice correspond to mutations in close proximity and with large cone of influence, while concepts close to the top might also combine distant mutations.

The definition of $(M, A_{\text{Blk}}, I_{\text{Blk}})$ can be refined in various ways, e.g. by 1. only considering certain classes of blocks in (4), for instance only relational operators; 2. disallowing to follow feedback connections (crossing unit-delay blocks) when checking reachability in (5); or 3. checking reachability not in the original model S, but in a k-fold unwinding of S. The generality of concept lattices gives rise to a systematic and very flexible method of analyzing sets of mutants.

4.2.3 Mutant selection using concept lattices

Given a concept lattice over mutants like the one defined in the previous section, step `(*)` of Alg. 1 can be implemented through traversal of the lattice, picking mutants from the visited concepts. Since mutants may occur in many

[2]For any set X, the expression X^* denotes the set of finite sequences of X-values.

concepts of the lattice, it is meaningful to restrict this selection to mutants that are specific for a chosen concept and do not occur in any sub-concepts:

$$N(X,Y) = \left\{ S' \in X \;\middle|\; \begin{array}{l} S' \notin X' \text{ for all concepts } (X',Y') \\ \text{with } (X',Y') < (X,Y) \end{array} \right\}$$

The set $N(X,Y)$ contains the mutants that occur in (X,Y), but not in any concept underneath (X,Y); intuitively, mutants in $N(X,Y)$ are those whose set of attributes coincides (or almost coincides) with the set Y. For the lattice in Fig. 2, for instance, we obtain $N(\{A,B,C\},\emptyset) = \emptyset$ and $N(\{C\},\{8,9\}) = \{C\}$.

To traverse the lattice over mutants, we can choose a linear order $(X_1,Y_1),\ldots,(X_n,Y_n)$ of the concepts in the lattice and implement `(*)` as:

> pick $S' \in N(X_i,Y_i) \cap M_r$, where $i \in \{1,\ldots,n\}$ is minimal such that $N(X_i,Y_i) \cap M_r \neq \emptyset$;

In our experiments, we consider four traversal orderings:

1. *Top-down breadth-first:* the order $C_1,\ldots,C_n$ determined by the distance of concepts from the top concept in the lattice.
2. *Top-down depth-first:* similarly, orders $C_1,\ldots,C_n$ determined by depth-first traversal of the lattice, starting from the top element.
3. *Bottom-up breadth-first:* the order given by the distance from the bottom element.
4. *Bottom-up depth-first:* the order given by depth-first traversal, starting from the bottom element.

It can be observed that the top-down strategies have similarity with MCTO from Sect. 3.2, while bottom-up strategies correspond to MFFO.

4.2.4 Mutant elimination using concept lattices

Besides ordering mutants, concept lattices can also be used to predict the outcome of the TCG process in Sect. 2.2 for a given mutant, thus making it possible to optimize step `(***)` in Alg. 1. This can be useful in two ways:

- If TCG has *failed* for one or for a small number of mutants selected from a concept (X,Y), it can be meaningful to ignore *all* mutants in X.[3]
- Vice versa, once a test-case has been generated for a mutant of the concept (X,Y), it can be meaningful to simply skip all further mutants in X, since it is likely that they are also killed by the new test-case.

In both scenarios, some amount of precision is traded off for performance, since predictions for further mutants of the selected concepts might be wrong. We quantify the loss of precision and the performance gains in Sect. 5. Both optimizations lead to the following implementation of step `(***)`:

> `//` (X,Y) `is the concept selected in (*)`
> $M_r \leftarrow M_r \setminus X$;
> identify further killed mutants in M_r by simulation;

[3]The TCG process can fail either because the mutant is in fact not observable, because of incompleteness of the TCG method, or simply because a timeout occurred. In each case, it is likely that TCG will also fail for other mutants in the same concept, so that a significant amount of time can be saved by skipping those mutants.

Benchmark	*#In*	*#Out*	*#Blks*	*#B-Blk*	*#Concepts*
CalcOffset	3	2	70	9	12
Decision	9	3	80	12	18
Safety	5	1	46	8	10
LocRecog	3	3	57	12	19
Sac	4	3	250	36	31
Quadratic	3	3	20	3	6
t-Sac-Safety	6	4	235	45	37
t-Sac-Safety-i	6	4	233	43	37

Table 2: Benchmark Characteristics

5. EXPERIMENTAL EVALUATION

We have implemented the test case generation algorithm and the mutant selection strategies described above. We use some of the tools presented in [2], namely a translator from Simulink to C, as well as the SAT-based bounded model checker CBMC.[4] This is combined with a mutation injection tool directly operating on Simulink models, as well as an implementation of the overall TCG procedure from Alg. 1. The computation of concept lattices is done using the ColibriConcepts[5] tool. All experiments are performed on a machine with a 3 GHz Intel Xeon CPU and 48 GB of memory running Linux.

Our experiments were conducted on eight Simulink models. For each model, we inject detectable and undetectable mutants. The benchmark named "Quadratic" is from [17]. The other benchmarks are extracted from Simulink models of embedded software provided by Ford. They contain control functions to implement steering anti catch-up. Table 2 summarizes some basic features of the models: *#In* and *#Out* denote the number of inputs and outputs, respectively, and *#Blks* is the number of blocks. The column *#B-Blk* gives the number of *branching blocks* in the model: when approximating the impact of each mutant, our tool focuses on certain types of blocks in the cone of influence of the mutation.[6] The rightmost column gives the number of concepts in the lattice using the traversal orderings 1) and 3) from Sec. 4.2. All but t-Sac-Safety-i use single precision floating-point arithmetic; the benchmark t-Sac-Safety-i is t-Sac-Safety rewritten to use integer arithmetic.

The runtime of each single TCG is limited to 20 minutes; the total time for a particular strategy is limited to five hours. The results are summarized in Table 3. The first column gives the name of the benchmark and the second the number of relevant mutants. The remaining columns present the results of applying the TCG algorithm to the benchmark using a particular selection heuristic. The columns "Cov" give the number of mutants covered and the columns "tc" give the number of test-cases required for this coverage. "Cov" marked with '*' represents the corresponding TCG procedure times out. The columns represent, respectively: top-down breadth-first traversal of the concept lattice, as in Sect. 4.2.3 (*Top-D*); bottom-up breadth-first traversal

[4]Available at `http://www.cprover.org/cover/`.
[5]`http://code.google.com/p/colibri-concepts/`
[6]These are the blocks *LogicOperator, MinMax, MultiportSwitch, RelationalOperator, Saturation, Switch*, and *Signum*. These block types are those typically considered for branch coverage by commercial tools for testing Simulink models, such as *Reactis Tester*.

Table 3: Experimental results

Benchmark	*#Muts*	*Top-D*		*Bot-U*		*Opt.*		*MFFO*		*MCTO*		*Rand-sel*		*Random*	
		Cov.	tc	Cov.	tc	Cov.	tc	Cov.	tc	Cov.	tc	Cov.	tc	Cov.	tc
CalcOffset	199	124	15	**125**	9	123	**6**	**125**	10	124	17	124	13	**125**	32
Decision	218	140	33	**141**	28	138	**12**	140	28	140	32	140	30	126	47
Safety	143	**114**	10	**114**	6	105	**5**	**114**	6	113	11	**114**	7	69	15
LocRecog	211	132	23	**135**	14	131	**7**	**135**	12	133	19	134	15	81	17
Sac	501	270*	30	**273***	21	271*	**20**	272*	27	251*	40	262*	23	206	16
Quadratic	66	**66**	11	**66**	9	**66**	9	**66**	**8**	**66**	12	**66**	10	**66**	17
t-Sac-Safety	621	163*	30	175*	18	**202**	**20**	171*	22	133*	15	172*	30	126	14
t-Sac-Safety-i	621	340*	35	**349***	25	339*	**10**	341*	30	335*	40	340*	33	113	25

(*Bot-U*); the mutant elimination strategy from Sect. 4.2.4 combined with bottom-up breadth-first traversal (*Opt*); two strategies from Sect. 3.2 selecting mutations w.r.t. observation points (*MFFO* and *MCTO*); random selection of mutants, as in Sect. 3.2 (*Rand-sel*). As a reference point, the last column (*Random*) gives the result of plain random testing, using 1000 randomly generated test vectors and a simple selection heuristic.

We observe that the two bottom-up concept lattice strategies (*Bot-U* and *Opt*) are overall able to achieve the highest degree of coverage, performing better or at least as good as all other approaches. The *Bot-U* strategy results in the best coverage within the timeout, while *Opt* usually generates the smallest number of test-cases. *Top-D* is dominated by *Bot-U*. The simple *MFFO* strategy is dominated by *Bot-U* with the exception of the "Quadratic" benchmark, where *MFFO* generates the smallest test suite. The strategy *MCTO* performs worse than random selection.

Comparing with the baseline approaches *Rand-sel* and *Random*, we observe that the coverage achieved by *Rand-sel* is comparable to that of *Top-D* and *MFFO*, but significantly worse than that of the bottom-up strategies *Bot-U* and *Opt*. With the exception of two benchmarks, pure random testing (*Random*) fails to achieve good coverage. Random testing usually produces a large number of test-cases; as example, random testing generates 32 test-cases for "CalcOffset" to kill 125 mutants, whereas *Bot-U* generates a suite of only 9 test-cases that cover the same number of mutants.

6. CONCLUSION

We present a new method for effectively applying an expensive TCG engine (a bounded model checker) to the problem of generating a small test suite that covers a large set of mutants of Simulink diagrams. We propose a novel ordering heuristic that is based on an change impact analysis using a formal concept lattice. Our experimental results indicate that the new method typically produces the smallest number of test cases and the best coverage.

7. REFERENCES

[1] D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina. Automatic test generation for coverage analysis using CBMC. In *Computer Aided Systems Theory (EUROCAST)*, volume 5717 of *LNCS*, pages 287–294. Springer, 2009.

[2] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, and G. Weissenbacher. Mutation-based test case generation for Simulink models. In *FMCO*, LNCS. Springer, 2010.

[3] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176. Springer, 2004.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[5] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34 –41, 1978.

[6] A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. C. Shashidar. AutoMOTGen: Automatic model oriented test generator for embedded control systems. In *CAV*, volume 5123 of *LNCS*, pages 204–208. Springer, 2008.

[7] B. Ganter and R. Wille. *Formal Concept Analysis*. Springer, 1996.

[8] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. FShell: Systematic test case generation for dynamic analysis and measurement. In *CAV*, volume 5123 of *LNCS*, pages 209–213. Springer, 2008.

[9] S. Hussain. *Mutation Clustering*. PhD thesis, King's College London, UK, 2008.

[10] N. Jha and S. Gupta. *Testing of Digital Systems*. Cambridge University Press, 2003.

[11] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)*, 2010.

[12] A. Kuehlmann and C. A. J. van Eijk. Combinational and sequential equivalence checking. In *Logic Synthesis and Verification*, pages 343–372. Kluwer, 2002.

[13] O. Kupferman, W. Li, and S. A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *FMCAD*, pages 1–9. IEEE, 2008.

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

[15] J. R. Ruthruff, M. M. Burnett, and G. Rothermel. Interactive fault localization techniques in a spreadsheet environment. *IEEE Transactions on Software Engineering (TSE)*, 32(4):213–239, 2006.

[16] P. V. Suman, T. Muske, P. Bokil, U. Shrotri, and R. Venkatesh. Masking boundary value coverage: Effectiveness and efficiency. In *TAIC PART*, volume 6303 of *LNCS*, pages 8–22. Springer, 2010.

[17] Y. Zhan and J. A. Clark. A search-based framework for automatic testing of MATLAB/Simulink models. *J. Syst. Softw.*, 81:262–285, February 2008.