# Tightening Test Coverage Metrics: A Case Study in Equivalence Checking using *k*-Induction[⋆]

Alastair F. Donaldson[1], Nannan He[1], Daniel Kroening[1], and Philipp Rümmer[2]

[1] Computer Science Department, Oxford University, UK
[2] Uppsala University, Department of Information Technology, Uppsala, Sweden

**Abstract.** We present a case study applying the $k$-induction method to equivalence checking of Simulink designs. In particular, we are interested in the problem of equivalence detection in mutation-based testing: given a design $S$, determining whether a "mutant" design $S'$ derived from $S$ by syntactic fault injection is behaviourally equivalent to $S$. In this situation, efficient equivalence checking techniques are needed to avoid redundant and expensive search for test cases that observe differences between $S$ and $S'$. We have integrated $k$-induction into our test case generation framework for Simulink. We show, using a selection of benchmarks, that $k$-induction can be effective in detecting equivalent mutants, sometimes as a stand-alone technique, and sometimes with some manual assistance. We further discuss how the level of automation of the method can be increased by using static analysis to derive strengthening invariants from the structure of the Simulink models.

## 1 Introduction

Mutation-based testing [20] is an effective technique for generating high-quality test suites for software systems. The technique is based on the hypothesis that a test capable of detecting a small, synthesized error in a program may very likely be able to detect real defects introduced accidentally by programmers. Applying mutation-based testing typically works as follows. A set of *mutations* is identified. These are small syntactic changes to the system under test. An example mutation might be the replacement of operator `+` by `*` at a particular program point, or the modification of a signal in a dataflow program by injecting a new computation block. Then, a search is carried out to find a set of test cases that *kill* many of the mutants: a test case kills a mutant if it exposes the fact that the behaviours of the original and mutated system diverge. The more mutants a test suite kills the higher the coverage of the suite, while the smaller the test suite the more efficiently it can be executed during software development. Given a sufficiently rich set of mutation operators, mutation coverage subsumes many

[⋆] This research is supported by the EU FP7 STREP MOGENTES (project ID ICT-216679), the ARTEMIS CESAR project, the EU FP7 STREP PINCETTE (project ID ICT-257647), and EPSRC grant EP/G051100.

other popular notions of coverage, such as location coverage and MC/DC for software and stuck-at faults for hardware [24].

Ideally, we would like to be able to efficiently derive a small set of test cases that kill all of a given set of mutants. A problem with this ideal is the possibility of *equivalent mutants*. Because mutants are obtained in a lightweight, syntactic fashion, there is no guarantee that a given mutant will in fact exhibit different behaviour from the original system. If no such difference can ever be observed, we say that that mutant is *equivalent* to the original system. Clearly, no test case can ever kill an equivalent mutant. We do not wish to waste time attempting to derive such test cases, and it would be unfair to regard a test suite to be of low quality because it does not kill mutants that are actually equivalent. Thus there is a need for techniques to automatically detect equivalence of mutants.

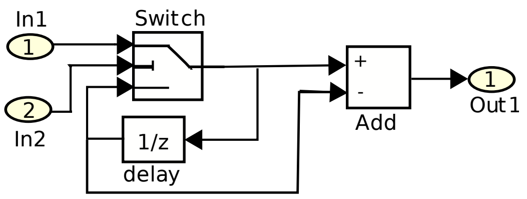
In previous work [5] we proposed a mutation-based test case generation approach for Simulink models. The basic idea is to inject multiple mutants into a Simulink model to obtain a mutated design. The original and mutated designs are automatically translated into an ANSI-C program that runs the designs in-step, asserting that they are observationally equivalent. Then, bounded model checking (BMC) [3] using a tool such as CBMC [6] is performed to check whether the designs really are observationally equivalent up to a given execution depth. Bounded equivalence is checked using a SAT solver like MiniSat, such that non-equivalence is detected when the solver finds a satisfying assignment to the associated SAT problem. Such satisfying assignments are used to derive test cases to kill the injected mutants.

While effective, this BMC-based approach is expensive, thus we would like to avoid applying it to injected mutants which are equivalent. In this work, we present a case study using the $k$-induction method [26] for equivalence checking in the Simulink domain. We show, using a selection of benchmarks, that $k$-induction can be effective in detecting equivalent mutants, sometimes as a stand-alone technique, and sometimes with some manual assistance. We further discuss how the level of automation of the method can be increased by using static analysis to derive strengthening invariants from the structure of the Simulink models.

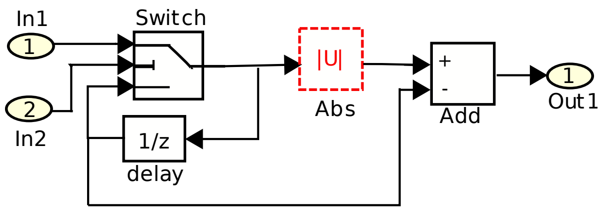
## 2 Mutation-based Test Case Generation for Simulink

### 2.1 Matlab Simulink

Matlab Simulink is a graphical dataflow language that is commonly used in industry 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



(a) Original Simulink model



(b) Mutated Simulink model

**Fig. 1.** Example of a mutated Simulink model

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 discrete-time Simulink models, which means that signals represent (potentially infinite) streams of values governed by a global clock. 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*).

An example of a Simulink model is given in Fig. 1(a). This model has two inports *In1*, *In2*, and one outport *Out1*. The inputs are connected to a *Switch* (a multiplexer) that forwards either *In1* or the output of *delay*, depending on the value of *In2*. The unit-delay block *delay* is responsible for storing the output of *Switch* for one time unit, thus preventing a cyclic definition caused otherwise by the feedback loop from *Switch* to itself. All instances of the unit-delay block must be initialized. The default initialization value is 0. The block *Add* (actually performing subtraction) computes the difference between the results of *Switch* and *delay* and feeds it to the outport *Out1*.

## 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 [5], 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*. We use $S'$ to refer to a mutant in $M$.

Fig. 1 is an example mutant derived from the Simulink model of Fig. 1(a). In the mutant, the output of *Switch* is replaced by its absolute value before being input to *Add*. Assume that an input has the form $(In1, In2)$, where *In2* causes the top input of *Switch* to be selected if it is 1, otherwise the bottom input is to be selected; and initial outputs for all unit delay blocks are 1. An example input sequence $\langle(1,1),(-1,1)\rangle$ leads to the output sequence of Fig. 1(a) as $\langle 0, \mathbf{-2}\rangle$, while applied to Fig. 1(b), the output sequence is $\langle 0, \mathbf{0}\rangle$ (recall that, as discussed in §2.1, the *Add* blocks of Fig. 1 actually perform subtraction). The difference in final values for these output sequences, highlighted in bold, indicates that the models behave differently.

The goal of TCG is to find a set of test cases (finite sequences of inputs for the model $S$) that *kill* each of the mutants in $M$, which means that the test case makes a mutant $S' \in M$ produce outputs that differ from those of the original model $S$. For example, as argued above, the test case $\{((1,1),(-1,1))\}$ kills the mutant of Fig. 1. 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* [9]).

In the style of bounded model checking [2], both the original model $S$ and each of its mutants $S' \in M$ can abstractly be modeled using transition relations $T$ and $T'$ and formulae $I$, $I'$ defining the initial states. Like in equivalence checking [21], observational equivalence of $S$ and $S'$ during the first $d$ computation steps can then be expressed using the following formula:

$$\underbrace{I(s_0) \wedge \bigwedge_{i=0}^{d-1} T(s_i, s_{i+1})}_{\text{first model}} \wedge \underbrace{I'(s'_0) \wedge \bigwedge_{i=0}^{d-1} T'(s'_i, s'_{i+1})}_{\text{second model}} \wedge \underbrace{\bigwedge_{i=0}^{d} s_i.i = s'_i.i}_{\text{equality of all inputs}} \Rightarrow \underbrace{\bigwedge_{i=0}^{d} s_i.o = s'_i.o}_{\text{equality of all outputs}} \qquad (1)$$

Any countermodel to 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 integers or floating-point arithmetic, and therefore have a finite state space, countermodels can be constructed using SAT/SMT-based techniques.

## 2.3 From Simulink to C: Our Test Case Generation Tool Chain

Due to the complexity of the Simulink language and the size of commonly used block type libraries, the development of analysis tools directly operating on Simulink models is a huge effort. We therefore follow a compilation approach and convert Simulink models to C programs prior to test case generation. Further processing can then be performed by ANSI-C analysis tools, in our case based on

```
// Declaration of inputs
signal_type in0, in1, ...;
// Declaration of internal signals
signal_type sig0, sig1, ..., sig0_m, sig1_m, ...;
// Declaration of outputs
signal_type out0, out1, ... out0_m, out1_m, ...;

int main () {
// The main simulation loop
  for (sim_time=START; sim_time<END; sim_time+=sim_step) {

// Reading inputs
    in0 = readInput0(); in1 = readInput1(); ...

// Execution of the original model S
    sig0 = ...;   sig1 = ...;    ...
    out0 = ...;   out1 = ...;    ...

// Execution of the mutant S'
    sig0_m = ...; sig1_m = ...; ...
    out0_m = ...; out1_m = ...; ...

// (*)
  }
}
```

**Fig. 2.** Skeleton of C code generated from Simulink models

the CBMC [6] bounded model checker. For the compilation from Simulink to C, we use two different tools: our own Simulink front-end [5], which is tightly integrated with CBMC and optimised for static analysis (applied to the generated C programs); and Gene-Auto [27], an industrial-grade open-source code generator for Simulink.

In our experiments, mutations are always applied at the level of Simulink models (rather than, as would also be possible, at the level of the C code generated from a Simulink model):

1. a given Simulink model $S$ is first duplicated (cloned) by creating a copy $S'$ of $S$ connected to the same input ports as $S$.
2. the clone $S'$ is mutated by inserting a further computation block into one of the signals of $S'$. The mutation operators considered in this paper are described in Sect. 5; the work presented here directly generalises to further mutation operators.

Examples of the resulting models are given in Fig. 5.

Compiling Simulink models to C results in programs of the structure shown in Fig. 2. The type `signal_type` will practically be either `int` or `float` (in our experiments, the former). Note that the program can contain multiple, nested loops in addition to the main simulation loop, since the code generated for the models $S$ and $S'$ might itself contain loops. Loops other than the main simulation loop are, however, usually bounded.

```
assert(out0 == out0_m && out1 == out1_m && ...);  // (*)
```

**Fig. 3.** Assertion relating the outputs of the original and mutated Simulink models. Counterexamples that violate this assertion provide test vectors that kill the mutant

In order to generate test cases, an assertion relating the different outputs is added at `(*)` in Fig. 2; the added assertion is shown in Fig. 3. Counterexamples demonstrating that the assertion `(*)` can be violated represent test cases killing the considered mutant. Such counterexamples can effectively be constructed using bounded model checkers such as CBMC [6]. The tool COVER [5] automates this process and produces test cases in an XML-based format.

### 2.4 The Phenomenon of Equivalent Mutants

Not all mutations give rise to observably different behaviour of a model. In fact, one of the main obstacles in traditional mutation testing is the difficulty of identifying mutations that do not have an observable effect on system outputs. Suppose formula (1) from §2.2 is valid for some $d$, which means that the applied mutation does not result in an error that propagates to an observable output within $d$ steps. There are two possible reasons for this:

1. The bound $d$ is not sufficiently large to reveal the error.
2. The model contains redundancy and the mutation does not result in an observable change of its behaviour. In other words, (1) is valid *for any* $d$. The mutant is in this case called an *equivalent mutant*.

The first case could be addressed by simply increasing the bound $d$. However, bounded model checking alone is not sufficient to distinguish between the two cases, since there is no upper bound (or only prohibitively large bounds, taking the usually finite state space of a Simulink program into account) on the values of $d$ that have to be considered. In order to detect case 2, it is therefore necessary to apply techniques beyond bounded model checking; the approach evaluated in this paper is based on strong versions of *induction* and inductive invariants.

## 3 Detection of Equivalent Mutants using $k$-Induction

The $k$-induction method was proposed as a technique for SAT-based verification of finite-state transition systems [26], and has been used successfully to verify complex hardware designs, in particular pipelined architectures. Recently, $k$-induction has also been applied in the verification of imperative software [12, 13]. In this paper, we consider applying the software formulation of $k$-induction proposed in [12, 13] to detect mutant equivalence in the C programs generated via the technique described in Sect. 2.3.

### 3.1 *k*-Induction for Transition Systems

Let $\mathbf{I}(x)$ and $\mathbf{T}(x, y)$ be formulae encoding the initial states and transition relation for a system over sets of state variables $x$ and $y$, $\mathbf{P}(x)$ a formula representing states satisfying a safety property, and $k$ a non-negative integer. To prove $\mathbf{P}$ by $k$-induction one must first show that $\mathbf{P}$ holds in all states reachable from an initial state within $k$ steps, i.e., that the following formula (the base case) is unsatisfiable:

$$\mathbf{I}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{T}(s_{k-1}, s_k) \wedge (\overline{\mathbf{P}(s_1)} \vee \cdots \vee \overline{\mathbf{P}(s_k)}) \tag{2}$$

Secondly, one must show that whenever $\mathbf{P}$ holds in $k$ consecutive states $s_1, \ldots, s_k$, $\mathbf{P}$ also holds in the next state $s_{k+1}$ of the system. This is established by checking that the following formula (the step case) is unsatisfiable:

$$\mathbf{P}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{P}(s_k) \wedge \mathbf{T}(s_k, s_{k+1}) \wedge \overline{\mathbf{P}(s_{k+1})} \tag{3}$$

Having proved both the base and step case, we can conclude that $\mathbf{P}$ holds in every reachable state of the transition system. The method can be made complete for finite-state systems by restricting the step case to consider only loop-free paths [26].

### 3.2 *k*-Induction in Mutation-Based Testing

In the context of mutation-based testing, we can use $k$-induction in order to show that (1) holds for any value of $d$. In this case, (1) forms the base case for $k$-induction: for some $d \geq 0$ we show that the original and mutated systems are equivalent up to depth $d$.

To obtain a complete result, we must also prove a step case as follows:

$$\begin{aligned} &\underbrace{\bigwedge_{i=0}^{d} T(s_i, s_{i+1})}_{\text{first model}} \wedge \underbrace{\bigwedge_{i=0}^{d} T'(s'_i, s'_{i+1})}_{\text{second model}} \wedge \underbrace{\bigwedge_{i=0}^{d} s_i.o = s'_i.o}_{\text{equality of first } d \text{ outputs}} \\ \Rightarrow\quad &\underbrace{s_{d+1}.o = s'_{d+1}.o}_{\text{equality of output } d+1} \end{aligned} \tag{4}$$

The step case ascertains that, given that the original and mutated systems have exhibited equal outputs for $d$ steps, they are guaranteed to show equivalent outputs for a further step.

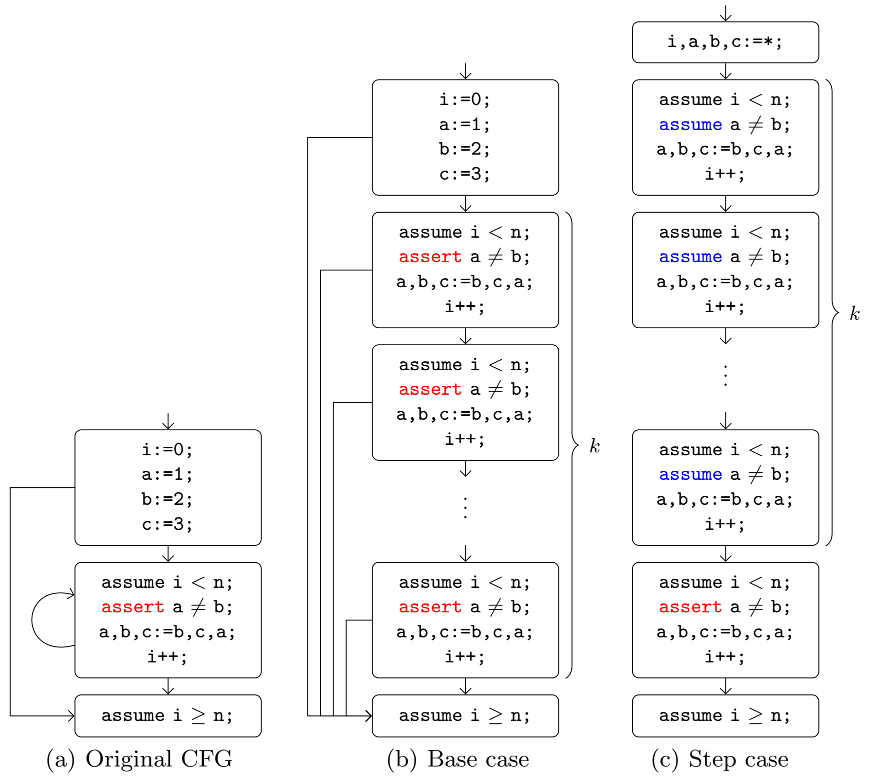


**Fig. 4.** A simple program, and the corresponding base and step cases for $k$-induction

## 3.3 $k$-Induction for Software Programs

In prior work [12, 13] we investigated a direct lifting of $k$-induction from transition systems to the level of program loops, in order to prove partial correctness of software programs with respect to assertions appearing in the program text.

Because we translate Simulink designs into C, it is this formulation of $k$-induction, rather than the transition system-level formulation outlined in §3.1 and §3.2, that we use to implement $k$-induction for detection of equivalent mutants. Our software $k$-induction method can directly be applied to programs such as the one shown in Fig. 2.

The formal definition of our $k$-induction rule for programs is quite complex, but the intuition is simple. We shall explain the idea using an example, referring the reader to [12, 13] for formal details.

We apply $k$-induction to a single loop in a program. Consider the example program of Fig. 4(a), depicted as a control flow graph (CFG), where flow of control is modelled using *assume* statements. (In particular, note that the loop condition $i < n$ is modelled by assuming that this expression holds on entry to the loop, and assuming that its negation holds on loop exit.)

We wish to prove that the assertion in the body of the loop can never be violated. This could be achieved using standard techniques by showing that $a \neq b \wedge a \neq c \wedge b \neq c$ is an inductive invariant for the loop, and that it implies the assertion $a \neq b$ of interest. However, with $k$-induction, we can prove this example correct *without* providing an external loop invariant. Instead, the assertion appearing in the loop body takes the role of an invariant.

From the CFG of Fig. 4(a), we derive two programs. The *step case program* (Fig. 4(c)) is analogous to (3). It checks whether, after executing the loop body successfully $k$ times from an arbitrary state, a further loop iteration can be successfully executed. In this further loop iteration, back edges to the loop header are removed, while edges that exit the loop are preserved. Thus the step case verifies that on loop exit, the rest of the program can be safely executed.

Because the program of Fig. 4(a) is indeed correct, the base case of Fig. 4(b) is correct for any $k \geq 0$. However, the step case of Fig. 4(c) is correct only for $k \geq 3$. To see that the step case does not hold for $k = 2$, consider the case where $n > 2$ and statement `i,a,b,c:=*` yields $i = 0$, $a = 1$, $b = 2$, $c = 1$. From this state, two loop iterations can be successfully executed, leading to a state where $a = 1$ , $b = 1$ and $c = 2$, at which point the assertion $a \neq b$ does not hold.

It is this program-level approach to $k$-induction which we employ in order to detect equivalence of mutants in Simulink designs, applying induction to the C programs generated from our compilation flow.

## 4 Automatic Invariant Strengthening

We shall see in §5 that naïve application of $k$-induction is not strong enough to show equivalence of mutants in some typical cases. The intuitive reason why $k$-induction might fail is that the asserted property—that the outputs generated by the original model $S$ and the mutant $S'$ are equal—is not $k$-inductive for any $k$, since the resulting induction hypothesis gives too little information about the internal state of the Simulink programs. In general, it can be necessary to strengthen the invariant by adding conditions about the range of signals, or by equations asserting that signals of the original model and of the mutants carry the same value.

We examine two techniques to strengthen invariants automatically: abstract interpretation, using numeric abstract domains, and van Eijk's method to infer equalities between signals. We evaluate both techniques experimentally in §5.

### 4.1 Abstract Interpretation

In prior work [10] we have investigated ways to strengthen $k$-induction through static analyses, including abstract interpretation [7]. Given a control-flow graph to be analysed, suppose we use abstract interpretation (with some suitable domain) to determine that an invariant $\phi$ holds on entry to node $n$. Then, because abstract interpretation is a sound method, we can prepend the statement *assume*($\phi$) to $n$. In practice, we choose to prepend the statement *assert*($\phi$)

rather than *assume*($\phi$). This forces $k$-induction to re-check the inferred invariants, guarding against the possibility of vacuous results arising from bugs in the abstract interpreter.

By exploiting information about invariants in this way, we increase the possibility for $k$-induction to succeed in proving the property of interest: while the property may not be $k$-inductive in general, it may be $k$-inductive when restricted to the invariant obtained using abstract interpretation.

In §5 we discuss mutants that cannot be proven equivalent using $k$-induction alone, but for which equivalence can be proven if abstract interpretation, over the domain of intervals, is first used to compute a strengthening invariant.

### 4.2 Adaptation of van Eijk's Method

The application of non-relational abstract domains (as in §4.1), for instance the interval domain, can significantly increase the proof strength of $k$-induction when detecting equivalent mutants. In this section, we propose a particular *relational* domain that further supports $k$-induction through eager computation of groups of signals that carry the same values in all executions of a Simulink model. The technique is inspired by van Eijk's method [15], a method for sequential equivalence checking of hardware designs. In the original version, the method works by computing classes of signals that have the same (or opposite) values in all reachable states of a circuit; the computation is done using BDDs (to describe equivalence classes of signals and the transition relation of the circuit) and fixed-point iteration. Van Eijk's method was combined with SAT-based verification, Stålmarck's method, and $k$-induction in [4].

The need for relational information is illustrated in Fig. 6(a), in which the equivalence of the considered mutant can only be shown when adding the additional assertion that the output of the unit-delay block `UD_m` is not affected by the applied mutation. This is done by asserting that the outputs of the blocks `UD` and `UD_m` are equal. In this example, the use of a non-relational abstract domain alone is not sufficient for the equivalence proof. This situation is typical for Simulink programs with internal state that cannot completely be observed at the program outputs.

In general, we assume that a relation

$$R \subseteq \underbrace{\{\texttt{sig0}, \texttt{sig1}, \ldots\}}_{Sig} \times \underbrace{\{\texttt{sig0\_m}, \texttt{sig1\_m}, \ldots\}}_{Sig^m}$$

between original and mutated internal signals has been identified (where variables are named as in Fig. 2). The assertion inserted at `(*)` in Fig. 2 is then strengthened to:

```
  assert(out0 == out0_m && out1 == out1_m && ...);     // (*)
{  assert(a == b); }(a,b)∈R
```

Let us call this stronger set of assertions $A(R)$. If it is possible to verify $A(R)$ using $k$-induction, then also the original assertion, and thus the equivalence of the mutant has been proven.

**Algorithm 1:** Iterative mutant equivalence checking

```
Input: C program as in Fig. 2, initial relation $R^c$, parameter $k \geq 0$
Output: One of {EQUIVALENT, NONEQUIVALENT, DONTKNOW}

$R \leftarrow R^c$;
/* Eliminate signal pair candidates using random simulation */
repeat
    Execute Fig. 2 with assertions $A(R)$ and random inputs;
    if assertion corresponding to pair $(a, b) \in R$ failed then
        $R \leftarrow R \setminus \{(a, b)\}$;
    else if difference in outputs observed then
        return NONEQUIVALENT;
    end
until timeout;

/* Check $k$-induction base case */
repeat
    Check $k$-induction base case with assertions $A(R)$;
    if assertion corresponding to pair $(a, b) \in R$ failed then
        $R \leftarrow R \setminus \{(a, b)\}$;
    else if difference in outputs observed then
        return NONEQUIVALENT;
    end
until base case succeeded;

/* Check $k$-induction step case */
repeat
    Check $k$-induction step case with assertions $A(R)$;
    if assertion corresponding to pair $(a, b) \in R$ failed then
        $R \leftarrow R \setminus \{(a, b)\}$;
    else if difference in outputs observed then
        return DONTKNOW;
    end
until step case succeeded;

return EQUIVALENT;
```

In order to automatically compute relations $R$ for which $k$-induction succeeds, we propose to first identify a set $R^c \subseteq Sig \times Sig^m$ of *candidate pairs* of signals. Natural candidates are pairs of corresponding signals in the original Simulink model and the mutant; such pairs are easy to compute and likely to carry the same values. Furthermore, the number of corresponding signal pairs is only linear in the size of the Simulink models. In some cases, it might, however, be beneficial to start from a larger set of signal pair candidates.

As second step, $R^c$ is refined to a set $R^i \subseteq R^c$ by removing signal pairs that can be shown to have different values in some executions. This is done using two methods:

– by trying to verify the $k$-induction base case for the set $A(R^c)$ of assertions. Most likely, such a verification attempt will initially fail and report that some of the assertions in $A(R^c)$ could not be verified; the corresponding pairs have to be removed from $R^i$.
– by random simulation of the program in Fig. 2, using the set $A(R^c)$ of assertions. In practice, random testing can be expected to efficiently and quickly remove large numbers of candidate pairs from $R^i$.

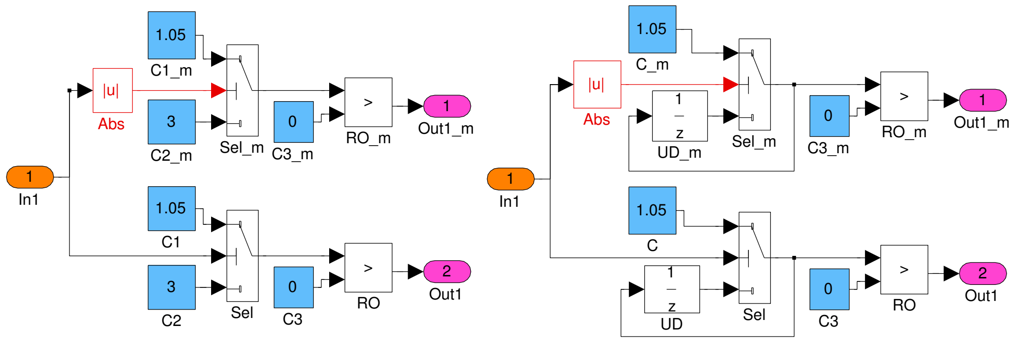
The set $R^i$ of remaining candidate pairs has to be refined by removing further signal pairs until the $k$-induction step case succeeds. This is done by trying to verify the $k$-induction step case with the set $A(R^i)$ of assertions; if some assertion of the step case cannot be verified, the corresponding pair $(a, b) \in R^i$ is removed, leading to the new relation $R^i := R^i \setminus \{(a, b)\}$. Iterating this procedure will eventually produce the greatest relation $R = R^i$ for which $k$-induction succeeds, or will terminate with the result that the equivalence of the considered mutant could not be proven.

Alg. 1 defines this technique more formally. The algorithm proceeds in three phases: 1. random simulation is used to remove as many signal pair candidates from $R$ as possible; 2. the $k$-induction base case is verified, potentially ruling out further signal pairs in $R$; and 3. the $k$-induction step case is verified, again reducing the set $R$ as needed. For a given timeout bound for random simulation, the algorithm is guaranteed to terminate, and outputs as result either that the considered mutant was proven to be equivalent, that the mutant is non-equivalent (in which case it is also possible to extract a test case killing the mutant from the algorithm), or that the equivalence check was inconclusive.

**Lemma 1 (Soundness of Alg. 1).** *If Alg. 1 returns the result* Equivalent *(*NonEquivalent*), the examined mutant is equivalent (not equivalent).*

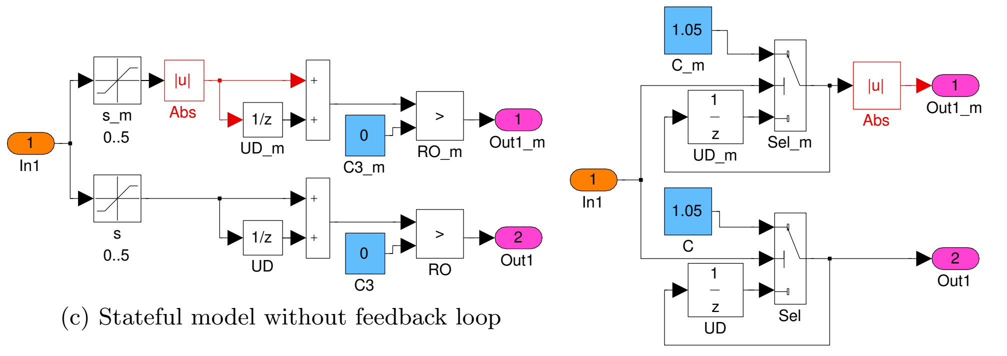
**Lemma 2 (Completeness of Alg. 1).** *If $R^c$ contains a sub-relation $R^s \subseteq R^c$ such that $k$-induction is able to verify the assertions $A(R^s)$ for the C program in Fig. 2, then Alg. 1 returns the result* Equivalent *when started with the initial relation $R^c$ and the parameter $k$.*

*Proof.* By showing the following two properties: 1. for all relations $R$ computed during the execution of the algorithm, it is the case that $R^s \subseteq R$; and 2. as long



(a) Stateless (combinational) model

(b) Model with feedback loop



(c) Stateful model without feedback loop

(d) Equivalence at the mutated location

**Fig. 5.** Basic mutation scenarios occurring in our Simulink benchmarks

as $R^s \subseteq R$ during the execution of the algorithm, the result DontKnow is not returned. □

It can be observed that Alg. 1 can be adapted also to other classes of assertions than just equations over signals; also the combination with the numeric abstract domains in Sect. 4.1 is straightforward.

## 5 Experiments

In this section, we first discuss four basic equivalent mutants extracted from larger, real-world benchmarks; we then report experimental results, and analyse two full-size examples in detail. To translate Simulink to C we use the tool presented in [5], as well as Gene-Auto [27]. For equivalence checking, we use K-Inductor [11], a prototypical version of Cbmc extended with $k$-induction. For our adaptation of van Eijk's method (§4.2), we have written a script which repeatedly invokes K-Inductor to check base and step cases.

All experiments are performed on a computer with a 3 GHz Intel Xeon CPU and 48 GB of memory, running Linux.

## 5.1 Simple Examples

In Fig. 5, each model consists of a mutant (the upper part) and the original model (the lower part). The mutant contains duplicates of all blocks in the original model; these blocks are distinguished by the suffix *_m*. The mutant and the original model share the same inports. The outports *Out1_m* and *Out1* are numbered 1 and 2 respectively, as they are distinct outports in the overall Simulink model. The red blocks (named *Abs*) are the applied mutations, and thus appear in the upper part of each model. These are inserted absolute-value blocks. Blue blocks (whose name starts with $C$) are constants, labelled with their respective values. All blocks labelled with '1/z' are *Unit Delays* (memory blocks) and initialised with the value 1.

It can be observed that the outputs generated by the mutants and the original models coincide, no matter what the inputs to the models are, so that the mutants are indeed equivalent.

In case (a), the model does not include any state-related blocks or feedback loops, and is thus purely combinational. Using K-Inductor, we could prove the equivalence of this mutant with $k = 1$.

Case (b) is more complex, since a *Unit Delay* block *UD* occurs after the mutated location. The input of this block is connected with the output of a *Switch* block *Sel*, carrying through either the upper or the lower input signal, depending on whether the value of the middle input is 0 or not. The output of *UD* is fed back to the switch *Sel*. This is a typical scenario observed in the benchmarks. Using $k$-induction alone, it was not possible to prove the equivalence of the mutant, since the step case could not be verified (for reasonable $k$). Verification is possible, however, when adding an assertion like `assert(Sel_m > 0 && Sel > 0)` into the generated C code, which makes the assertions in the simulation loop $k$-inductive for $k = 2$. Abstract interpretation over intervals, as discussed in §4.1, allows this assertion to be automatically derived.

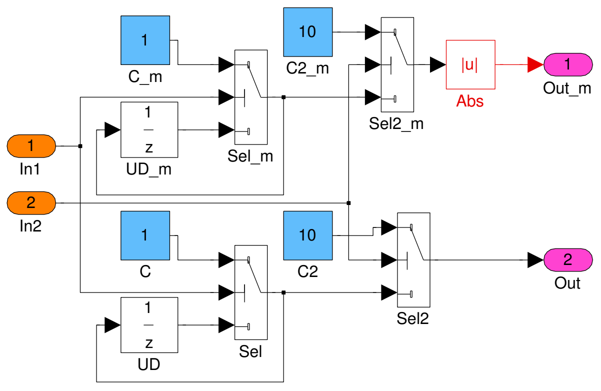
Case (c) also includes a *Unit Delay* block, but is simpler because the output of the block *UD* is not fed back to its input. In this case, equivalence can directly be proven using $k$-induction with $k = 2$.

Case (d) is a special case where the mutation directly affects the output of the model. Also in this example, equivalence can directly be proven using $k$-induction with $k = 2$.

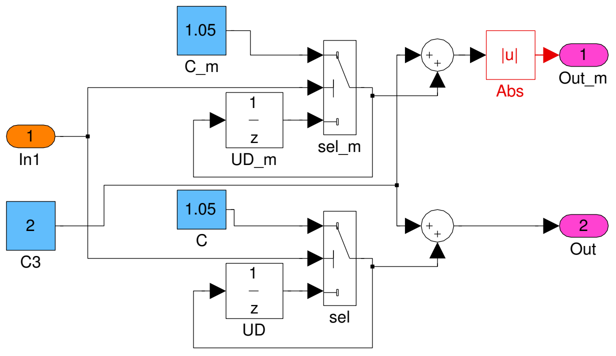
Two more interesting variants of this case are given in Fig. 6. We observe that $k$-Induction can directly prove Fig. 6(b), but not (a). In order to verify equivalence in (a), it is necessary to add two further assertions to the simulation loop:

```
assert(UD_M == UD); assert(UD_M >= 0);
```

To obtain the first invariant, the adaptation of van Eijk's method presented in §4.2 is used: the relation $R^c$ is initialised with all pairs of signal variables occurring between the mutation point and outports. Algorithm 1 is then applied to remove invalid pairs. We did not employ random simulation, the first phase of



(a) Nested switches



(b) Sum after switch

**Fig. 6.** Two cases of equivalences at mutated location

Algorithm 1, to eliminate invalid pairs; we leave this to future work. Currently, elimination is performed solely by checking the $k$-induction base and step cases.

The second invariant can be derived automatically using abstract interpretation over intervals as discussed in §4.1.

## 5.2 Larger Simulink Case Studies

In this case study, we make use of $k$-induction to check whether mutants injected into Simulink models are equivalent. We consider four Simulink benchmarks extracted from an industrial embedded software system used in the European MOGENTES project.[3] This software system contains control functions to implement steering anti catch-up in an automobile. We applied three different mutation operators to the Simulink models:

– ABS: Insert absolute value.

[3] `https://www.mogentes.eu/`

**Table 1.** Summary of experimental results

| *Benchmark* | *#Blocks* | *#Muts* | *#S-blocks* | *#Muts-RandT* | *#K-ind* | *#K-ind-IS* | *#Ineq* |
|---|---|---|---|---|---|---|---|
| CalcOffset | 80 | 69 | 2 | 10 | 3 | 3 | 7 |
| Decision | 92 | 87 | 2 | 15 | 0 | 7 | 8 |
| RecogLoc | 66 | 40 | 1 | 11 | 0 | 5 | 6 |
| Spoiler | 44 | 36 | 0 | 5 | 5 | 5 | 0 |

– UOI: Insert negation $(-, \neg)$ operator.
– RR: Swap relational operators $<, \leq, >, \geq, =$.

The results of proving equivalence of mutants directly using $k$-induction, or after strengthening with added invariant assertions are summarised in Table 1. For every benchmark, *#Blocks* reports the total number of Simulink blocks (of the original model), while *#Muts* shows the number of generated mutants, each including exactly one mutation. *#S-blocks* gives the number of state-related blocks in the model (for example *Unit-Delay* blocks).

The *#Muts-RandT* column shows the number of mutants left after applying a simple random testing approach to kill mutants. *#K-Ind* and *#K-Ind-IS* give the number of mutants that are proved equivalent using our $k$-induction-based technique without and with invariant strengthening respectively (where strengthening uses abstract interpretation and our adaptation of van Eijk's method). Note that the number for *#K-Ind-IS* is always larger than that for *#K-Ind*, because invariant strengthening can only increase the proof strength of our method.

The last column *#Ineq* reports the number of mutants which were not proved inequivalent using random simulation, but which were shown inequivalent by the base case of our $k$-induction approach. (Essentially, this means that these mutants can be shown to be inequivalent using shallow bounded model checking.)

In all cases, we find that (*#K-ind-IS* + *#Ineq*) = (*#Muts-RandT*), thus our technique is able to fully categorise mutants as equivalent or inequivalent for this set of benchmarks.

The *CalcOffset* benchmark includes two *Unit Delay* blocks. The outputs of these blocks are not fed back to their inputs, similar to Fig. 5(c). The experiments show that $k$-induction alone is capable of proving equivalent mutants, and the runtime of each proof is within 1 second. For the benchmarks *Decision* and *RecogLoc*, invariant strengthening indeed increases the number of mutants that $k$-induction can prove equivalent. The *Spoiler* benchmark model has no state-related blocks (similar to Fig. 5(a)); all potentially equivalent mutants are indeed proven equivalent by $k$-induction alone within 1 second.

Table 2 gives detailed information about mutants injected into the *Decision* and *RecogLoc* models. As shown in Table 1, none of these mutants could be shown equivalent by $k$-induction alone—all required invariant strengthening. The column *#T(s)* gives the runtime needed for verification, in seconds. It includes the total time for running our abstract interpreter, followed by the van Eijk-based method of Algorithm 1 if needed (which may call K-Inductor multiple times). The column $k$ gives the parameter $k$ required to prove equivalence

**Table 2.** Experimental results of proved mutants

| *ID* | *Benchmark* | *Mutation-Type* | *#T(s)* | *k* | *#Assert. v.E.* |
|---|---|---|---|---|---|
| M-1 | Decision | ABS | 1.21 | 2 | N/A |
| M-2 | Decision | ABS | 3.23 | 2 | 3/3 |
| M-3 | Decision | ABS | 4.15 | 2 | 5/5 |
| M-4 | Decision | ABS | 4.34 | 4 | 6/1 |
| M-5 | Decision | ABS | 1.23 | 3 | N/A |
| M-6 | Decision | ABS | 3.24 | 3 | 6/4 |
| M-7 | Decision | RR | 3.72 | 3 | 5/1 |
| M-8 | RecogLoc | ABS | 3.81 | 4 | 5/2 |
| M-9 | RecogLoc | ABS | 3.78 | 4 | 4/2 |
| M-10 | RecogLoc | ABS | 4.74 | 4 | 6/4 |
| M-11 | RecogLoc | ABS | 3.31 | 2 | 6/4 |
| M-12 | RecogLoc | RR | 3.51 | 2 | 6/4 |

of a mutant. The last column shows the number of signal-equivalence invariant assertions which were added by the van Eijk-based method of Algorithm 1, if this strengthening is necessary. For example, for the mutant *M-1*, "N/A" means that $k$-induction strengthened with abstract interpretation is sufficient to prove the equivalence, without invoking our adaptation of van Eijk's method. In contrast, for mutant *M-6*, "6/4" reports that six potential assertions are identified by the van Eijk technique, and four of them are proved to be invariants and used in strengthening $k$-induction. Overall, the experiments show that most of these equivalent mutants (10 out of 12) require strengthening with both abstract interpretation and our adaptation of van Eijk's method in order for $k$-induction to succeed. We did not find any cases among the *Decision* or *RecogLoc* mutants where the van Eijk technique allowed equivalence to be proven without abstract interpretation also being applied.

## 6 Related Work

*Mutation-based test case generation.* The concept of mutation testing was first introduced in 1971 in Richard Lipton's class term paper "Fault diagnosis of computer programs." Since then, it has become a standard method for evaluating the quality of test suites and been applied to software systems of considerable size, see [20] for a broad survey. In this paper, we only consider mutant models with single mutations, whereas other authors also consider combinations of faults [22]. In [25], Ruthermel et al. propose to use mutations to prioritise test cases to increase a test suite's rate of fault detection.

This work is partly based on our framework [5] for test case generation for Simulink. An optimisation of the framework, applying formal concept analysis to cluster mutants, has been described in [19].

*$k$-Induction.* The concept of $k$-induction was first published in [26, 4], targeting the verification of hardware designs represented by transition relations (although

the basic idea had already been used in earlier implementations [23] and a version of one-induction used for BDD-based model checking [8]). A major emphasis of these two papers is on the restriction to loop-free or shortest paths, which is so far not considered in our $k$-induction rule due to the size of state vectors and the high degree of determinism in software programs. Several optimisations and extensions to the technique have been proposed, including property strengthening to reduce induction depth [28], improving performance via incremental SAT solving [14], and verification of temporal properties [1].

Besides hardware verification, $k$-induction has been used to analyse synchronous programs [18, 16], SystemC designs [17] and imperative software [12, 13]. A combination of the $k$-induction rule of [12, 13], abstract interpretation, and domain-specific invariant strengthening techniques for race analysis is the topic of [10].

## 7 Conclusions and Future Work

We have presented a case study in the application of $k$-induction to the problem of detecting equivalent mutants in mutation-based test case generation for Matlab Simulink. Our experiments show that $k$-induction shows promise in this area, proving successful in equivalence detection for a range of examples, sometimes as a stand-alone technique, but often requiring assistance from other static analyses: abstract interpretation over intervals, and an adaptation of van Eijk's method for sequential equivalence checking. The experiments also show that strong versions of induction, such as $k$-induction with $k > 1$, are beneficial for equivalence proofs: several of our example proofs could only be conducted with $k \geq 2$.

Future work will involve completing the implementation of the van Eijk-based method by using random simulation to eliminate invalid pairs of signal variables, and analysing the effectiveness of our method over a larger class of Simulink designs.

## References

1. Armoni, R., Fix, L., Fraer, R., Huddleston, S., Piterman, N., Vardi, M.Y.: SAT-based induction for temporal safety properties. Electr. Notes Theor. Comput. Sci. 119(2), 3–16 (2005)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 193–207. Springer (1999)
4. Bjesse, P., Claessen, K.: SAT-based verification without state space traversal. In: Formal Methods in Computer-Aided Design (FMCAD). pp. 372–389. Springer (2000)

5. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for Simulink models. In: Formal Methods for Components and Objects (FMCO). LNCS, vol. 6286, pp. 208–227. Springer (2009)
6. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 168–176. Springer (2004)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (POPL). pp. 238–252. ACM (1977)
8. Déharbe, D., Moreira, A.M.: Using induction and BDDs to model check invariants. In: CHARME. IFIP Conference Proceedings, vol. 105, pp. 203–213. Chapman & Hall (1997)
9. DeMillo, R., Lipton, R., Sayward, F.: Hints on test data selection: Help for the practicing programmer. Computer 11(4), 34 –41 (April 1978)
10. Donaldson, A.F., Haller, L., Kroening, D.: Strengthening induction-based race checking with lightweight static analysis. In: Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, Springer (2011)
11. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Proceedings of the 18th International Static Analysis Symposium (SAS'11). Lecture Notes in Computer Science, Springer (2011), to appear
12. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 6015, pp. 280–295. Springer (2010)
13. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of DMA races using model checking and $k$-induction. Formal Methods in System Design (2011)
14. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4) (2003)
15. van Eijk, C.A.J.: Sequential equivalence checking without state space traversal. In: Proceedings of the conference on Design, automation and test in Europe (DATE). pp. 618–623. IEEE (1998)
16. Franzén, A.: Using satisfiability modulo theories for inductive verification of Lustre programs. Electr. Notes Theor. Comput. Sci. 144(1), 19–33 (2006)
17. Große, D., Le, H.M., Drechsler, R.: Proving transaction and system-level properties of untimed SystemC TLM designs. In: MEMOCODE. pp. 113–122. IEEE Computer Society (2010)
18. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: FMCAD. pp. 109–117. IEEE (2008)
19. He, N., Rümmer, P., Kroening, D.: Test-case generation for embedded Simulink via formal concept analysis. In: Proceedings of DAC (2011)
20. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering (TSE) (2010)
21. Kuehlmann, A., van Eijk, C.A.J.: Combinational and sequential equivalence checking. In: Logic Synthesis and Verification, pp. 343–372. Kluwer International Series in Engineering and Computer Science Series, Kluwer (2002)
22. 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). pp. 1–9. IEEE (2008)
23. Lillieroth, C.J., Singh, S.: Formal verification of FPGA cores. Nord. J. Comput. 6(3), 299–319 (1999)

24. Offutt, J., Voas, J.M.: Subsumption of condition coverage techniques by mutation testing. Tech. Rep. ISSE-TR-96-01, George Mason University (1996)
25. Ruthruff, J.R., Burnett, M.M., Rothermel, G.: Interactive fault localization techniques in a spreadsheet environment. IEEE Transactions on Software Engineering (TSE) 32(4), 213–239 (2006)
26. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Formal Methods in Computer-Aided Design (FMCAD). LNCS, vol. 1954, pp. 108–125. Springer (2000)
27. Toom, A., Izerrouken, N., Naks, T., Pantel, M., Kai, O.S.Y.: Towards reliable code generation with an open tool: Evolutions of the Gene-Auto toolset. In: Proceedings, Embedded Real Time Software and Systems (ERTS) (2010)
28. Vimjam, V.C., Hsiao, M.S.: Explicit safety property strengthening in SAT-based induction. In: VLSID. pp. 63–68. IEEE (2007)