Mutation Testing with Hyperproperties[⋆]

Andreas Fellner [1,2], Mitra Tabaei Befrouei [2], and Georg Weissenbacher[2]

[1] AIT Austrian Institute of Technology, [2] TU Wien

Abstract. We present a new method for model-based mutation-driven test case generation. Mutants are generated by making small syntactical modifications to the model or source code of the system under test. A test case kills a mutant if the behavior of the mutant deviates from the original system when running the test. In this work, we use hyperproperties—which allow to express relations between multiple executions—to formalize different notions of *killing* for both deterministic as well as non-deterministic models. The resulting hyperproperties are universal in the sense that they apply to arbitrary reactive models and mutants. Moreover, an off-the-shelf model checking tool for hyperproperties can be used to generate test cases. We evaluate our approach on a number of models expressed in two different modeling languages by generating tests using a state-of-the-art mutation testing tool.

1 Introduction

Mutations—small syntactic modifications of programs that mimic typical programming errors—are used to assess the quality of existing test suites. A test *kills* a mutated program (or *mutant*), obtained by applying a *mutation operator* to a program, if its outcome for the mutant deviates from the outcome for the unmodified program. The percentage of mutants killed by a given test suite serves as a metric for test quality. The approach is based on two assumptions: (a) the *competent programmer hypothesis* [11], which states that implementations are typically close-to-correct, and (b) the *coupling effect* [27], which states that a test suites ability to detect simple errors (and mutations) is indicative of its ability to detect complex errors.

In the context of model-based testing, mutations are also used to design tests. Model-based test case generation is the process of deriving tests from a reference model (which is assumed to be free of faults) in such a way that they reveal any non-conformance of the reference model and its mutants, i.e., kill the mutants. The tests detect potential errors (modeled by mutation operators) of implementations, treated as a black box in this setting, that conform to a mutant instead of the reference model. A test *strongly* kills a mutant if it triggers an observable difference in behavior [11], and *weakly* kills a mutant if the deviation is merely in a difference in traversed program states [22].

The aim of our work is to automatically construct tests that strongly kill mutants derived from a reference model. To this end, we present two main contributions:

[⋆] The research was supported by ECSEL JU under the project H2020 737469 AutoDrive — Advancing failaware, fail-safe, and fail-operational electronic components, systems, and architectures for fully automated driving to make future mobility safer, affordable, and end-user acceptable, by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005, and by the Austrian National Research Network S11403-N23 (RiSE)

(1) A formalization of mutation killing in terms of *hyperproperties* [14], a formalism to relate multiple execution traces of a program which has recently gained popularity due to its ability to express security properties such as non-interference and observational determinism. Notably, our formalization also takes into account potential non-determinism, which significantly complicates killing of mutants due to the unpredictability of the test outcome.
(2) An approach that enables the automated construction of tests by means of *model checking* the proposed hyperproperties on a model that aggregates the reference model and a mutant of it. To circumvent limitations of currently available model checking tools for hyperproperties, we present a transformation that enables the control of non-determinism via additional program inputs. We evaluate our approach using a state-of-the-art model checker on a number of models expressed in two different modeling languages.

Running example. We illustrate the main concepts of our work in Figure 1. Figure 1a shows the SMV [25] model of a beverage machine, which non-deterministically serves `coff` (coffee) or `tea` after input `req` (request), assuming that there is still enough `wtr` (water) in the tank. Water can be refilled with input `fill`. The symbol ε represents absence of input and output, respectively.

The code in Figure 1a includes the variable `mut` (initialized non-deterministically in line 4), which enables the activation of a mutation in line 10. The mutant refills 1 unit of water only, whereas the original model fills 2 units.

Figure 1b states a hyperproperty over the inputs and outputs of the model formalizing that the mutant can be killed *definitely* (i.e., independently of non-deterministic choices). The execution shown in Figure 1c is a witness for this claim: the test requests two drinks after filling the tank. For the mutant, the second request will necessarily fail, as indicated in Figure 1d, which shows all possible output sequences for the given test.

Outline. Section 2 introduces our system model and HyperLTL. Section 3 explains the notions of *potential* and *definite* killing of mutants, which are then formalized in terms of hyperproperties for deterministic and non-deterministic models in Section 4. Section 5 introduces a transformation to control non-determinism in models, and Section 6 describes our experimental results. Related work is discussed in Section 7.

2 Preliminaries

This section introduces symbolic transition systems as our formalisms for representing discrete reactive systems and provides the syntax and semantics of HyperLTL, a logic for hyperproperties.

2.1 System Model

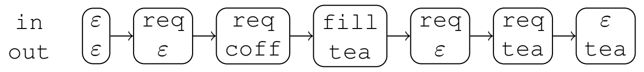
A symbolic transition system (STS) is a tuple $\mathcal{S} = \langle \mathcal{I}, \mathcal{O}, \mathcal{X}, \alpha, \delta \rangle$, where $\mathcal{I}, \mathcal{O}, \mathcal{X}$ are finite sets of input, output, and state variables, α is a formula over $\mathcal{X} \cup \mathcal{O}$ (the initial

```
init(in)  :=ε
init(out):=ε
init(wtr):=2
init(mut):={⊤,⊥}
next(in)  :={ε,req, fill}
next(out):=
  if(in=req&wtr>0):{coff, tea}
  else              :ε
next(wtr):=
  if   (in=fill):(mut ? 1 : 2)
  elif(in=req&wtr>0):wtr−1
  else                   :wtr
next(mut):=mut
```

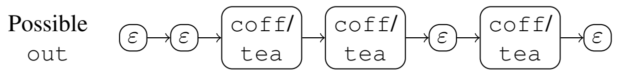
(a) Beverage machine with cond. mutant

$$
\begin{aligned}
&\exists\pi\forall\pi'\forall\pi''\\
&\Box\big(\neg\texttt{mut}_{\pi} \wedge \texttt{mut}_{\pi'} \wedge \neg\texttt{mut}_{\pi''} \wedge\\
&\quad([\texttt{in=}\varepsilon]_{\pi} \leftrightarrow [\texttt{in=}\varepsilon]_{\pi'} \leftrightarrow [\texttt{in=}\varepsilon]_{\pi''})\wedge\\
&\quad([\texttt{in=req}]_{\pi} \leftrightarrow [\texttt{in=req}]_{\pi'}\\
&\qquad\qquad\leftrightarrow [\texttt{in=req}]_{\pi''})\wedge\\
&\quad([\texttt{in=fill}]_{\pi} \leftrightarrow [\texttt{in=fill}]_{\pi'}\\
&\qquad\qquad\leftrightarrow [\texttt{in=fill}]_{\pi''})\big) \rightarrow\\
&\Diamond\big(\neg([\texttt{o=}\varepsilon]_{\pi'} \leftrightarrow [\texttt{o=}\varepsilon]_{\pi''})\vee\\
&\quad\neg([\texttt{o=coff}]_{\pi'} \leftrightarrow [\texttt{o=coff}]_{\pi''})\vee\\
&\quad\neg([\texttt{o=tea}]_{\pi'} \leftrightarrow [\texttt{o=tea}]_{\pi''})\big)
\end{aligned}
$$

(b) Hyperproperty expressing killing



(c) Definitely killing test



(d) Spurious test response of mutant

Fig. 1: Beverage machine running example

conditions predicate), and δ is a formula over $\mathcal{I} \cup \mathcal{O} \cup \mathcal{X} \cup \mathcal{X}'$ (the transition relation predicate), where $\mathcal{X}' = \{x' \mid x \in \mathcal{X}\}$ is a set of primed variables representing the successor states. An input I, output O, state X, and successor state X', respectively, is a mapping of $\mathcal{I}, \mathcal{O}, \mathcal{X}$, and $\mathcal{X}'$, respectively, to values in a fixed domain that includes the elements $\top$ and $\bot$ (representing true and false, respectively). $Y|_{\mathcal{V}}$ denotes the restriction of the domain of mapping Y to the variables $\mathcal{V}$. Given a valuation Y and a Boolean variable $v \in \mathcal{V}$, $Y(v)$ denotes the value of v in Y (if defined) and $Y[v]$ and $Y[\neg v]$ denote Y with v set to $\top$ and $\bot$, respectively.

We assume that the initial conditions- and transition relation predicate are defined in a logic that includes standard Boolean operators $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$. We omit further details, as as our results do not depend on a specific formalism. We write $X, O \models \alpha$ and $I, O, X, X' \models \delta$ to denote that α and δ evaluate to true under an evaluation of inputs I, outputs O, states X, and successor states X'. We assume that every STS has a distinct output O_ε, representing absence of output.

A state X with output O such that $X, O \models \alpha$ are an *initial state* and *initial output*. A state X has a transition with input I to its *successor state* X' with output O iff $I, O, X, X' \models \delta$, denoted by $X \xrightarrow{I,O} X'$. A *trace* of $\mathcal{S}$ is a sequence of tuples of concrete inputs, outputs, and states $\langle(I_0, O_0, X_0), (I_1, O_1, X_1), (I_2, O_2, X_2), \ldots\rangle$ such that $X_0, O_0 \models \alpha$ and $\forall j \geq 0 \,.\, X_j \xrightarrow{I_j, O_{j+1}} X_{j+1}$. We require that every state has at least one successor, therefore all traces of $\mathcal{S}$ are infinite. We denote by $\mathcal{T}(\mathcal{S})$ the set of all traces of $\mathcal{S}$. Given a trace $p = \langle (I_0, O_0, X_0), (I_1, O_1, X_1), \ldots \rangle$, we write $p[j]$ for (I_j, O_j, X_j), $p[j, l]$ for $\langle (I_j, O_j, X_j), \ldots, (I_l, O_l, X_l) \rangle$, $p[j, \infty]$ for $\langle (I_j, O_j, X_j), \ldots \rangle$ and $p|_{\mathcal{V}}$ to denote $\langle (I_0|_{\mathcal{V}}, O_0|_{\mathcal{V}}, X_0|_{\mathcal{V}}), (I_1|_{\mathcal{V}}, O_1|_{\mathcal{V}}, X_1|_{\mathcal{V}}), \ldots \rangle$. We lift restriction to sets of traces T by defining $T|_{\mathcal{V}}$ as $\{p|_{\mathcal{V}} \mid t \in T\}$.

$\mathcal{S}$ is *deterministic* iff there is a unique pair of an initial state and initial output and for each state X and input I, there is at most one state X' with output O, such that $X \xrightarrow{I,O} X'$. Otherwise, the model is *non-deterministic*.

In the following, we presume the existence of sets of atomic propositions $\mathsf{AP} = \{\mathsf{AP}_{\mathcal{I}} \cup \mathsf{AP}_{\mathcal{O}} \cup \mathsf{AP}_{\mathcal{X}}\}$ (intentionally kept abstract)[1] serving as labels that characterize inputs, outputs, and states (or properties thereof).

For a trace $p = \langle (I_0, O_0, X_0), (I_1, O_1, X_1), \ldots \rangle$ the corresponding trace over AP is $\mathsf{AP}(p) = \langle \mathsf{AP}(I_0) \cup \mathsf{AP}(O_0) \cup \mathsf{AP}(X_0), \mathsf{AP}(I_1) \cup \mathsf{AP}(O_1) \cup \mathsf{AP}(X_1), \ldots \rangle$. We lift this definition to sets of traces by defining $\mathsf{APTr}(\mathcal{S}) \overset{\text{def}}{=} \{\mathsf{AP}(p) \mid p \in \mathcal{T}(\mathcal{S})\}$.

Example 1. Figure 1a shows the formalization of a beverage machine in SMV [25]. In Figure 1b, we use atomic propositions to enumerate the possible values of `in` and `out`. This SMV model closely corresponds to an STS: the initial condition predicate α and transition relation δ are formalized using integer arithmetic as follows:

$$
\begin{aligned}
\alpha \overset{\text{def}}{=} & \texttt{out=}\varepsilon \wedge \texttt{wtr=2} \\
\delta \overset{\text{def}}{=} & \texttt{wtr>0} \wedge \texttt{in=req} \wedge \texttt{out=coff} \wedge \texttt{wtr'=wtr-1} \vee \\
& \texttt{wtr>0} \wedge \texttt{in=req} \wedge \texttt{out=tea} \wedge \texttt{wtr'=wtr-1} \vee \\
& \texttt{in=fill} \wedge \neg\texttt{mut} \wedge \texttt{out=}\varepsilon \wedge \texttt{wtr'=2} \vee \\
& \texttt{in=fill} \wedge \texttt{mut} \wedge \texttt{out=}\varepsilon \wedge \texttt{wtr'=1} \vee \\
& \texttt{in=}\varepsilon \wedge \texttt{out=}\varepsilon \wedge \texttt{wtr'=wtr}
\end{aligned}
$$

The trace $p = \langle (\varepsilon, \varepsilon, 2), (\texttt{req}, \varepsilon, 2), (\texttt{req}, \texttt{coff}, 1), (\varepsilon, \texttt{tea}, 0), \ldots \rangle$ is one possible execution of the system (for brevity, variable names are omitted). Examples of atomic propositions for the system are $[\texttt{in=coff}], [\texttt{out=}\varepsilon], [\texttt{wtr>0}], [\texttt{wtr=0}]$ and the respective atomic proposition trace of p is $\mathsf{AP}(p) = \langle \{[\texttt{in=}\varepsilon], [\texttt{out=}\varepsilon], [\texttt{wtr>0}]\}, \{[\texttt{in=req}], [\texttt{out=}\varepsilon], [\texttt{wtr>0}]\}, \{[\texttt{in=req}], [\texttt{out=coff}], [\texttt{wtr>0}]\}, \{[\texttt{in=req}], [\texttt{out=tea}], [\texttt{wtr=0}]\} \ldots \rangle$

2.2 HyperLTL

In the following, we provide an overview of the HyperLTL, a logic for hyperproperties, sufficient for understanding the formalization in Section 4. For details, we refer the reader to [13]. HyperLTL is defined over atomic proposition traces (see Section 2.1) of a fixed STS $\mathcal{S} = \langle \mathcal{I}, \mathcal{O}, \mathcal{X}, \alpha, \delta \rangle$ as defined in Section 2.1.

[1] Finite domains can be characterized using binary encodings; infinite domains require an extension of our formalism in Section 2.2 with equality and is omitted for the sake of simplicity.

Syntax. Let AP be a set of atomic propositions and let π be a *trace variable* from a set $\mathcal{V}$ of trace variables. Formulas of HyperLTL are defined by the following grammar:

$$\begin{aligned} \psi &::= \exists \pi.\psi \mid \forall \pi.\psi \mid \quad \varphi \\ \varphi &::= \quad a_\pi \quad \mid \ \neg\varphi \ \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi \end{aligned}$$

Connectives $\exists$ and $\forall$ are universal and existential trace quantifiers, read as "along some traces" and "along all traces". In our setting, atomic propositions $a \in \mathsf{AP}$ express facts about states or the presence of inputs and outputs. Each atomic proposition is sub-scripted with a trace variable to indicate the trace it is associated with. The Boolean connectives $\wedge$, $\rightarrow$, and $\leftrightarrow$ are defined in terms of $\neg$ and $\vee$ as usual. Furthermore, we use the standard temporal operators *eventually* $\Diamond\varphi \overset{\text{def}}{=} \mathsf{true}\ \mathcal{U}\varphi$, and *always* $\Box\varphi \overset{\text{def}}{=} \neg\Diamond\neg\varphi$.

Semantics $\Pi \models_{\mathcal{S}} \psi$ states that ψ is valid for a given mapping $\Pi : \mathcal{V} \rightarrow \mathsf{APTr}(\mathcal{S})$ of trace variables to atomic proposition traces. Let $\Pi\,[\pi \mapsto p]$ be as Π except that π is mapped to p. We use $\Pi\,[i,\infty]$ to denote the trace assignment $\Pi'(\pi) = \Pi(\pi)\,[i,\infty]$ for all π. The validity of a formula is defined as follows:

$$\begin{array}{lll} \Pi \models_{\mathcal{S}} a_\pi & \text{iff} & a \in \Pi(\pi)[0] \\ \Pi \models_{\mathcal{S}} \exists\pi.\psi & \text{iff} & \text{there exists } p \in \mathsf{APTr}(\mathcal{S}) : \Pi\,[\pi \mapsto p] \models_{\mathcal{S}} \psi \\ \Pi \models_{\mathcal{S}} \forall\pi.\psi & \text{iff} & \text{for all } p \in \mathsf{APTr}(\mathcal{S}) : \Pi\,[\pi \mapsto p] \models_{\mathcal{S}} \psi \\ \Pi \models_{\mathcal{S}} \neg\varphi & \text{iff} & \Pi \not\models_{\mathcal{S}} \varphi \\ \Pi \models_{\mathcal{S}} \psi_1 \vee \psi_2 & \text{iff} & \Pi \models_{\mathcal{S}} \psi_1 \text{ or } \Pi \models_{\mathcal{S}} \psi_2 \\ \Pi \models_{\mathcal{S}} \bigcirc\varphi & \text{iff} & \Pi\,[1,\infty] \models_{\mathcal{S}} \varphi \\ \Pi \models_{\mathcal{S}} \varphi_1 \mathcal{U} \varphi_2 & \text{iff} & \text{there exists } i \geq 0 : \Pi\,[i,\infty] \models_{\mathcal{S}} \varphi_2 \\ & & \text{and for all } 0 \leq j < i \text{ we have } \Pi\,[j,\infty] \models_{\mathcal{S}} \varphi_1 \end{array}$$

We write $\models_{\mathcal{S}} \psi$ if $\Pi \models_{\mathcal{S}} \psi$ holds and Π is empty. We call $q \in \mathcal{T}(\mathcal{S})$ a π-witness of a formula $\exists\pi.\psi$, if $\Pi\,[\pi \mapsto p] \models_{\mathcal{S}} \psi$ and $\mathsf{AP}(q) = p$.

3 Killing mutants

In this section, we introduce mutants, tests, and the notions of potential and definite killing. We discuss how to represent an STS and its corresponding mutant as a single STS, which can then be model checked to determine killability.

3.1 Mutants

Mutants are variations of a model $\mathcal{S}$ obtained by applying small modifications to the syntactic representation of $\mathcal{S}$. A mutant of an STS $\mathcal{S} = \langle \mathcal{I}, \mathcal{O}, \mathcal{X}, \alpha, \delta \rangle$ (the *original model*) is an STS $\mathcal{S}^m = \langle \mathcal{I}, \mathcal{O}, \mathcal{X}, \alpha^m, \delta^m \rangle$ with equal sets of input, output, and state variables as $\mathcal{S}$ but a deviating initial predicate and/or transition relation. We assume that $\mathcal{S}^m$ is equally input-enabled as $\mathcal{S}$, that is $\mathcal{T}(\mathcal{S}^m)|_{\mathcal{I}} = \mathcal{T}(\mathcal{S})|_{\mathcal{I}}$, i.e., the mutant and model accept the same sequences of inputs. In practice, this can easily be achieved by using self-loops with empty output to ignore unspecified inputs. We use standard mutation operators, such as disabling transitions, replacing operators, etc. Due to space limitations and the fact that mutation operators are not the primary focus of this work, we do not list them here, but refer to the Appendix of [16] and [5]. We combine an original model represented by $\mathcal{S}$ and a mutant $\mathcal{S}^m$ into a *conditional mutant* $\mathcal{S}^{c(m)}$, in order to perform mutation analysis via model checking the combined model.

The conditional mutant is defined as $\mathcal{S}^{c(m)} \overset{\text{def}}{=} \langle \mathcal{I}, \mathcal{O}, \mathcal{X} \cup \{\text{mut}\}, \alpha^{c(m)}, \delta^{c(m)} \rangle$, where mut is a fresh Boolean variable used to distinguish states of the original and the mutated STS.

Suppose $\mathcal{S}^m$ replaces a sub-formula δ_0 of δ by δ_0^m, then the transition relation predicate of the conditional mutant $\delta^{c(m)}$ is obtained by replacing δ_0 in δ by $(\text{mut} \wedge \delta_0^m) \vee (\neg\text{mut} \wedge \delta_0)$. We fix the value of mut in transitions by conjoining δ with $\text{mut} \leftrightarrow \text{mut}'$. The initial conditions predicate of the conditional mutant is defined similarly.

Consequently, for a trace $p \in \mathcal{T}(\mathcal{S}^{c(m)})$ it holds that if $p|_{\{\text{mut}\}} = \{\bot\}^\omega$ then $p|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{T}(\mathcal{S})$, and if $p|_{\{\text{mut}\}} = \{\top\}^\omega$ then $p|_{\mathcal{I}\cup\mathcal{O}\cup\mathcal{X}} \in \mathcal{T}(\mathcal{S}^m)$. Formally, $\mathcal{S}^{c(m)}$ is non-deterministic, since mut is chosen non-deterministically in the initial state. However, we only refer to $\mathcal{S}^{c(m)}$ as non-deterministic if either $\mathcal{S}$ or $\mathcal{S}^m$ is non-deterministic, as mut is typically fixed in the hypertproperties presented in Section 4.

Example 1 and Figure 1a show a conditional mutant as an STS and in SMV.

3.2 Killing

Killing a mutant amounts to finding inputs for which the mutant produces outputs that deviate from the original model. In a reactive, model-based setting, killing has been formalized using conformance relations [29], for example in [4,15], where an implementation *conforms* to its specification if all its input/output sequences are part of/allowed by the specification.

In model-based testing, the model takes the role of the specification and is assumed to be correct by design. The implementation is treated as black box, and therefore mutants of the specification serve as its proxy. Tests (i.e., input/output sequences) that demonstrate non-conformance between the model and its mutant can be used to check whether the implementation adheres to the specification or contains the bug reflected in the mutant. The execution of a test on a system under test fails if the sequence of inputs of the test triggers a sequence of outputs that deviates from those predicted by the test. Formally, tests are defined as follows:

Definition 1 (Test). *A* test t *of* length n *for* $\mathcal{S}$ *comprises inputs* $t|_\mathcal{I}$ *and outputs* $t|_\mathcal{O}$ *of length* n*, such that there exists a trace* $p \in \mathcal{T}(\mathcal{S})$ *with* $p|_\mathcal{I}[0,n] = t|_\mathcal{I}$ *and* $p|_\mathcal{O}[0,n] = t|_\mathcal{O}$.

For non-deterministic models, in which a single sequence of inputs can trigger different sequences of outputs, we consider two different notions of killing. We say that a mutant can be *potentially killed* if there exist inputs for which the mutant's outputs deviate from the original model given an appropriate choice of non-deterministic initial states and transitions. In practice, executing a test that potentially kills a mutant on a faulty implementation that exhibits non-determinism (e.g., a multi-threaded program) may fail to demonstrate non-conformance (unless the non-determinism can be controlled). A mutant can be *definitely killed* if there exists a sequence of inputs for which the behaviors of the mutant and the original model deviate independently of how non-determinism is resolved.

Note potential and definite killability are orthogonal to the folklore notions of weak and strong killing, which capture different degrees of observability. Formally, we define potential and definite killability as follows:

Definition 2 (Potentially killable). *$\mathcal{S}^m$ is* potentially killable *if*

$$\mathcal{T}(\mathcal{S}^m)|_{\mathcal{I}\cup\mathcal{O}} \not\subseteq \mathcal{T}(\mathcal{S})|_{\mathcal{I}\cup\mathcal{O}}$$

Test t for $\mathcal{S}$ of length n potentially kills *$\mathcal{S}^m$ if*

$$\{q[0,n] \mid q \in \mathcal{T}(\mathcal{S}^m) \wedge q[0,n]|_{\mathcal{I}} = t|_{\mathcal{I}}\}|_{\mathcal{I}\cup\mathcal{O}} \not\subseteq \{p[0,n] \mid p \in \mathcal{T}(\mathcal{S})\}|_{\mathcal{I}\cup\mathcal{O}}.$$

Definition 3 (Definitely killable). *$\mathcal{S}^m$ is* definitely killable *if there is a sequence of inputs $\vec{I} \in \mathcal{T}(\mathcal{S})|_{\mathcal{I}}$, such that*

$$\{q \in \mathcal{T}(\mathcal{S}^m) \mid q|_{\mathcal{I}} = \vec{I}\}|_{\mathcal{O}} \cap \{p \in \mathcal{T}(\mathcal{S}) \mid p|_{\mathcal{I}} = \vec{I}\}|_{\mathcal{O}} = \emptyset$$

Test t for $\mathcal{S}$ of length n definitely kills *$\mathcal{S}^m$ if*

$$\begin{gathered}\{q[0,n] \mid q \in \mathcal{T}(\mathcal{S}^m) \wedge q[0,n]|_{\mathcal{I}} = t|_{\mathcal{I}}\}|_{\mathcal{O}} \cap \\ \{p[0,n] \mid p \in \mathcal{T}(\mathcal{S}) \wedge p[0,n]|_{\mathcal{I}} = t|_{\mathcal{I}}\}|_{\mathcal{O}} = \emptyset\end{gathered}$$

Definition 4 (Equivalent Mutant). *$\mathcal{S}^m$ is* equivalent *iff $\mathcal{S}^m$ is not potentially killable.*

Note that definite killability is stronger than potential killabilty, though for deterministic systems, the two notions coincide.

Proposition 1. *If $\mathcal{S}^m$ is definitely killable then $\mathcal{S}^m$ is potentially killable.*
If $\mathcal{S}^m$ is deterministic then: $\mathcal{S}^m$ is potentially killable iff $\mathcal{S}^m$ is definitely killable.

The following example shows a definitely killable mutant, a mutant that is only potentially killable, and an equivalent mutant.

Example 2. The mutant in Figure 1a, is definitely killable, since we can force the system into a state in which both possible outputs of the original system (`coff`, `tea`) differ from the only possible output of the mutant (ε).

Consider a mutant that introduces non-determinism by replacing line 7 with the code `**if**(in=fill):(mut ? {1,2} : 2)`, indicating that the machine is filled with either 1 or 2 units of water. This mutant is potentially but not definitely killable, as only one of the non-deterministic choices leads to a deviation of the outputs.

Finally, consider a mutant that replaces line 4 with `**if**(in=req&wtr>0):(mut ? coff : {coff,tea})` and removes the `mut` branch of line 7, yielding a machine that always creates coffee. Every implementation of this mutant is also correct with respect to the original model. Hence, we consider the mutant equivalent, even though the original model, unlike the mutant, can output `tea`.

4 Killing with hyperproperties

In this section, we provide a formalization of potential and definite killability in terms of HyperLTL, assert the correctness of our formalization with respect to Section 3, and explain how tests can be extracted by model checking the HyperLTL properties. All HyperLTL formulas depend on inputs and outputs of the model, but are model-agnostic otherwise. The idea of all presented formulas is to discriminate between traces of the original model ($\square\neg\mathrm{mut}_\pi$) and traces of the mutant ($\square\mathrm{mut}_\pi$). Furthermore, we quantify over pairs (π, π') of traces with globally equal inputs ($\square \bigwedge_{i\in\mathsf{AP}_\mathcal{I}} i_\pi \leftrightarrow i_{\pi'}$) and express that such pairs will eventually have different outputs ($\lozenge \bigvee_{o\in\mathsf{AP}_\mathcal{O}} \neg(o_\pi \leftrightarrow o_{\pi'})$).

4.1 Deterministic Case

To express killability (potential and definite) of a deterministic model and mutant, we need to find a trace of the model ($\exists\pi$) such that the trace of the mutant with the same inputs ($\exists\pi'$) eventually diverges in outputs, formalized by ϕ_1 as follows:

$$\phi_1(\mathcal{I},\mathcal{O}) := \exists\pi\exists\pi'\square(\neg\mathrm{mut}_\pi \wedge \mathrm{mut}_{\pi'} \bigwedge_{i\in\mathsf{AP}_\mathcal{I}} i_\pi \leftrightarrow i_{\pi'}) \wedge \lozenge(\bigvee_{o\in\mathsf{AP}_\mathcal{O}} \neg(o_\pi \leftrightarrow o_{\pi'}))$$

Proposition 2. *For a deterministic model $\mathcal{S}$ and mutant $\mathcal{S}^m$ it holds that*

$$\mathcal{S}^{c(m)} \models \phi_1(\mathcal{I},\mathcal{O}) \text{ iff } \mathcal{S}^m \text{ is killable.}$$

If t is a π-witness for $\mathcal{S}^{c(m)} \models \phi_1(\mathcal{I},\mathcal{O})$, then $t[0,n]|_{\mathcal{I}\cup\mathcal{O}}$ kills $\mathcal{S}^m$ (for some $n \in \mathbb{N}$).

4.2 Non-deterministic Case

For potential killability of non-deterministic models and mutants,[2] we need to find a trace of the mutant ($\exists\pi$) such that all traces of the model with the same inputs ($\forall\pi'$) eventually diverge in outputs, expressed in ϕ_2:

$$\phi_2(\mathcal{I},\mathcal{O}) := \exists\pi\forall\pi'\square(\mathrm{mut}_\pi \wedge \neg\mathrm{mut}_{\pi'} \bigwedge_{i\in\mathsf{AP}_\mathcal{I}} i_\pi \leftrightarrow i_{\pi'}) \rightarrow \lozenge(\bigvee_{o\in\mathsf{AP}_\mathcal{O}} \neg(o_\pi \leftrightarrow o_{\pi'}))$$

Proposition 3. *For non-deterministic $\mathcal{S}$ and $\mathcal{S}^m$, it holds that*

$$\mathcal{S}^{c(m)} \models \phi_2(\mathcal{I},\mathcal{O}) \text{ iff } \mathcal{S}^m \text{ is potentially killable.}$$

If s is a π-witness for $\mathcal{S}^{c(m)} \models \phi_2(\mathcal{I},\mathcal{O})$, then for any trace $t \in \mathcal{T}(\mathcal{S})$ with $t|_\mathcal{I} = s|_\mathcal{I}$, $t[0,n]|_{\mathcal{I}\cup\mathcal{O}}$ potentially kills $\mathcal{S}^m$ (for some $n \in \mathbb{N}$).

[2] The Appendix of [16] covers deterministic models with non-deterministic mutants and vice-versa.

To express definite killability, we need to find a sequence of inputs of the model ($\exists\pi$) and compare all non-deterministic outcomes of the model ($\forall\pi'$) to all non-deterministic outcomes of the mutant ($\forall\pi''$) for these inputs, as formalized by ϕ_3:

$$\phi_3(\mathcal{I},\mathcal{O}) := \exists\pi\forall\pi'\forall\pi''\Box\big(\neg\mathrm{mut}_\pi \wedge \mathrm{mut}_{\pi'} \wedge \neg\mathrm{mut}_{\pi''} \wedge \\ \bigwedge_{i\in\mathsf{AP}_\mathcal{I}} i_\pi \leftrightarrow i_{\pi'} \wedge i_\pi \leftrightarrow i_{\pi''}\big) \rightarrow \Diamond\big(\bigvee_{o\in\mathsf{AP}_\mathcal{O}} \neg(o_{\pi'} \leftrightarrow o_{\pi''})\big)$$

In Figure 1b, we present an instance of ϕ_3 for our running example.

Proposition 4. *For non-deterministic $\mathcal{S}$ and $\mathcal{S}^m$, it holds that*

$$\mathcal{S}^{c(m)} \models \phi_3(\mathcal{I},\mathcal{O}) \textit{ iff } \mathcal{S}^m \textit{ is definitely killable.}$$

If t is a π-witness for $\mathcal{S}^{c(m)} \models \phi_3(\mathcal{I},\mathcal{O})$, then $t[0,n]|_{\mathcal{I}\cup\mathcal{O}}$ definitely kills $\mathcal{S}^m$ (for some $n \in \mathbb{N}$).

To generate tests, we use model checking to verify whether the conditional mutant satisfies the appropriate HyperLTL formula presented above and obtain test cases as finite prefixes of witnesses for satisfaction.

5 Non-deterministic models in practice

As stated above, checking the validity of the hyperproperties in Section 4 for a given model and mutant enables test-case generation. To the best of our knowledge, MCHYPER [18] is the only currently available HyperLTL model checker. Unfortunately, MCHYPER is unable to model check formulas with alternating quantifiers.[3] Therefore, we are currently limited to checking $\phi_1(\mathcal{I},\mathcal{O})$ for deterministic models, since witnesses of ϕ_1 may not satisfy ϕ_2 in the presence of non-determinism.

To remedy this issue, we propose a transformation that makes non-determinism *controllable* by means of additional inputs and yields a deterministic STS. The transformed model over-approximates killability in the sense that the resulting test cases only kill the original mutant if non-determinism can also be controlled in the system under test. However, if equivalence can be established for the transformed model, then the original non-deterministic mutant is also equivalent.

5.1 Controlling non-determinism in STS

The essential idea of our transformation is to introduce a fresh input variable that enables the control of non-deterministic choices in the conditional mutant $\mathcal{S}^{c(m)}$. The new input is used carefully to ensure that choices are consistent for the model and the mutant encoded in $\mathcal{S}^{c(m)}$. W.l.o.g., we introduce an input variable nd with a domain sufficiently large to encode the non-deterministic choices in $\alpha^{c(m)}$ and $\delta^{c(m)}$, and write

[3] While *satisfiability* in the presence of quantifier alternation is supported to some extent [17].

$nd(X, O)$ to denote a value of nd that uniquely corresponds to state X with output O. Moreover, we add a fresh Boolean variable x^τ to $\mathcal{X}$ used to encode a fresh initial state.

Let $\mathcal{X}_+ \overset{\text{def}}{=} \mathcal{X} \cup \{\text{mut}\}$ and X_+, X'_+, I, O be valuations of $\mathcal{X}_+$, $\mathcal{X}'_+$, $\mathcal{I}$, and $\mathcal{O}$, and X and X' denote $X_+|_\mathcal{X}$ and $X'_+|_{\mathcal{X}'}$, respectively. Furthermore, $\psi(X)$, $\psi(X_+, I)$, and $\psi(O, X'_+)$ are formulas uniquely satisfied by X, (X_+, I), and (O, X'_+) respectively.

Given conditional mutant $\mathcal{S}^{c(m)} \overset{\text{def}}{=} \langle \mathcal{I}, \mathcal{O}, \mathcal{X}_+, \alpha^{c(m)}, \delta^{c(m)} \rangle$, we define its controllable counterpart $D(\mathcal{S}^{c(m)}) \overset{\text{def}}{=} \langle \mathcal{I} \cup \{nd\}, \mathcal{O}, \mathcal{X}_+ \cup \{x^\tau\}, D(\alpha^{c(m)}), D(\delta^{c(m)}) \rangle$. We initialize $D(\delta^{c(m)}) \overset{\text{def}}{=} \delta^{c(m)}$ and incrementally add constraints as described below.

Non-deterministic initial conditions: Let X be an arbitrary, fixed state. The unique fresh initial state is $X^\tau \overset{\text{def}}{=} X[x^\tau]$, which, together with an empty output, we enforce by the new initial conditions predicate:

$$D(\alpha^{c(m)}) \overset{\text{def}}{=} \psi(X^\tau, O_\varepsilon)$$

We add the conjunct $\neg\psi(X^\tau) \rightarrow \neg x^{\tau\prime}$ to $D(\delta^{c(m)})$, in order to force x^τ evaluating to $\bot$ in all states other than X^τ. In addition, we add transitions from X^τ to all pairs of initial states/outputs in $\alpha^{c(m)}$. To this end, we first partition the pairs in $\alpha^{c(m)}$ into pairs shared by and exclusive to the model and the mutant:

$$\begin{aligned}
J^\cap &\overset{\text{def}}{=} \{(O, X_+) \mid X, O \models \alpha^{c(m)}\} \\
J^{orig} &\overset{\text{def}}{=} \{(O, X_+) \mid \neg X_+(\text{mut}) \wedge (X_+, O \models \alpha^{c(m)}) \wedge (X_+[\text{mut}], O \not\models \alpha^{c(m)})\} \\
J^{mut} &\overset{\text{def}}{=} \{(O, X_+) \mid X_+(\text{mut}) \wedge (X_+, O \models \alpha^{c(m)}) \wedge (X_+[\neg\text{mut}], O \not\models \alpha^{c(m)})\}
\end{aligned}$$

For each $(O, X_+) \in J^\cap \cup J^{mut} \cup J^{orig}$, we add the following conjunct to $D(\delta^{c(m)})$:

$$\psi(X^\tau) \wedge nd(O, X) \rightarrow \psi(O, X'_+)$$

In addition, for inputs $nd(O, X)$ without corresponding target state in the model or mutant, we add conjuncts to $D(\delta^{c(m)})$ that represent self loops with empty outputs:

$$\begin{aligned}
\forall (O, X_+) \in J^{orig} &: \psi(X^\tau[\text{mut}]) \wedge nd(O, X) \rightarrow \psi(O_\varepsilon, X^{\tau\prime}[\text{mut}]) \\
\forall (O, X_+) \in J^{mut} &: \psi(X^\tau[\neg\text{mut}]) \wedge nd(O, X) \rightarrow \psi(O_\varepsilon, X^{\tau\prime}[\neg\text{mut}])
\end{aligned}$$

Non-deterministic transitions: Analogous to initial states, for each state/input pair, we partition the successors into successors shared or exclusive to model or mutant:

$$\begin{aligned}
T^\cap_{(X_+, I)} &\overset{\text{def}}{=} \{(X_+, I, O, X'_+) \mid X \xrightarrow{I,O} X'\} \\
T^{orig}_{(X_+, I)} &\overset{\text{def}}{=} \{(X_+, I, O, X'_+) \mid \neg X_+(\text{mut}) \wedge (X_+ \xrightarrow{I,O} X'_+) \wedge \neg(X_+[\text{mut}] \xrightarrow{I,O} X'_+)\} \\
T^{mut}_{(X_+, I)} &\overset{\text{def}}{=} \{(X_+, I, O, X'_+) \mid X_+(\text{mut}) \wedge (X_+ \xrightarrow{I,O} X'_+) \wedge \neg(X_+[\neg\text{mut}] \xrightarrow{I,O} X'_+)\}
\end{aligned}$$

A pair (X_+, I) causes non-determinism if

$$|(T^\cap_{(X_+, I)} \cup T^{orig}_{(X_+, I)})|_{\mathcal{X} \cup \mathcal{I} \cup \mathcal{O} \cup \mathcal{X}'}| > 1 \text{ or } |(T^\cap_{(X_+, I)} \cup T^{mut}_{(X_+, I)})|_{\mathcal{X} \cup \mathcal{I} \cup \mathcal{O} \cup \mathcal{X}'}| > 1.$$

For each pair (X_+, I) that causes non-determinism and each $(X_+, I, O_j, X'_{+j}) \in T^{\cap}_{(X_+,I)} \cup T^{mut}_{(X_+,I)} \cup T^{orig}_{(X_+,I)}$, we add the following conjunct to $D(\delta^{c(m)})$:

$$\psi(X_+, I) \land nd(O_j, X_j) \to \psi(O_j, X'_{+j})$$

Finally, we add conjuncts representing self loops with empty output for inputs that have no corresponding transition in the model or mutant:

$$\forall (X_+, I, O_j, X'_{+j}) \in T^{orig}_{(X_+,I)} : \psi(X_+[\text{mut}], I) \land nd(O_j, X_j) \to \psi(O_\varepsilon, X'_{+j}[\text{mut}])$$
$$\forall (X_+, I, O_j, X'_{+j}) \in T^{mut}_{(X_+,I)} : \psi(X_+[\neg\text{mut}], I) \land nd(O_j, X_j) \to \psi(O_\varepsilon, X'_{+j}[\neg\text{mut}])$$

The proposed transformation has the following properties:

Proposition 5. *Let $\mathcal{S}$ be a model with inputs $\mathcal{I}$, outputs $\mathcal{O}$, and mutant $\mathcal{S}^m$ then*

1. *$D(\mathcal{S}^{c(m)})$ is deterministic (up to* mut*).*
2. *$\mathcal{T}(\mathcal{S}^{c(m)})|_{\mathcal{X}_+ \cup \mathcal{I} \cup \mathcal{O}} \subseteq \mathcal{T}(D(\mathcal{S}^{c(m)}))[1, \infty]|_{\mathcal{X}_+ \cup \mathcal{I} \cup \mathcal{O}}$.*
3. *$D(\mathcal{S}^{c(m)}) \not\models \phi_1(\mathcal{I}, \mathcal{O})$ then $\mathcal{S}^m$ is equivalent.*

The transformed model is deterministic, since we enforce unique initial valuations and make non-deterministic transitions controllable through input nd. Since we only add transitions or augment existing transitions with input nd, every transition $X \xrightarrow{I,O} X'$ of $\mathcal{S}^{c(m)}$ is still present in $D(\mathcal{S}^{c(m)})$ (when input nd is disregarded). The potential additional traces of Item 2 originate from the O_ε-labeled transitions for non-deterministic choices present exclusively in the model or mutant. These transitions enable the detection of discrepancies between model and mutant caused by the introduction or elimination of non-determinism by the mutation.

For Item 3 (which is a direct consequence of Item 2), assume that the original non-deterministic mutant is not equivalent (i.e., potentially killable). Then $D(\mathcal{S}^{c(m)}) \models \phi_1(\mathcal{I}, \mathcal{O})$, and the corresponding witness yields a test which kills the mutant assuming non-determinism can be controlled in the system under test. Killability purported by ϕ_1, however, could be an artifact of the transformation: determinization potentially deprives the model of its ability to match the output of the mutant by deliberately choosing a certain non-deterministic transition. In Example 2, we present an equivalent mutant which is killable after the transformation, since we will detect the deviating output `tea` of the model and ε of the mutant. Therefore, our transformation merely allows us to provide a lower bound for the number of equivalent non-deterministic mutants.

5.2 Controlling non-determinism in modeling languages

The exhaustive enumeration of states (J) and transitions (T) outlined in Section 5.1 is purely theoretical and infeasible in practice. However, an analogous result can often be achieved by modifying the syntactic constructs of the underlying modeling language that introduce non-determinism, namely:

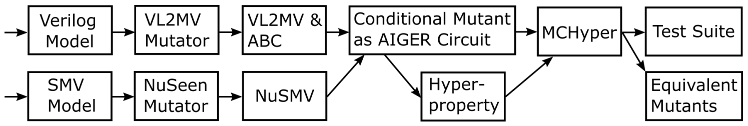


Fig. 2: Tool Pipeline of our Experiments

- *Non-deterministic assignments.* Non-deterministic choice over a finite set of elements $\{x'_1, \ldots x'_n\}$, as provided by SMV [25], can readily be converted into a case-switch construct over nd. More generally, explicit non-deterministic assignments $\mathtt{x} := \star$ to state variables $\mathtt{x}$ [26] can be controlled by assigning the value of nd to $\mathtt{x}$.
- *Non-deterministic schedulers.* Non-determinism introduced by concurrency can be controlled by introducing input variables that control the scheduler (as proposed in [23] for bounded context switches).

In case non-determinism arises through variables under-specified in transition relations, these variable values can be made inputs as suggested by Section 5.1. In general, however, identifying under-specified variables automatically is non-trivial.

Example 3. Consider again the SMV code in Figure 1a, for which non-determinism can be made controllable by replacing line **`if`**`(in=req&wtr>0):{coff,tea}` with lines **`if`**`(nd=0&in=req&wtr>0):coff`, **`elif`**`(nd=1&in=req&wtr>0):tea` and adding **`init`**`(nd):={0,1}`.

Similarly, the STS representation of the beverage machine, given in Example 1, can be transformed by replacing the first two rules by the following two rules:

$$\mathtt{nd=0} \wedge \mathtt{wtr>0} \wedge \mathtt{in=req} \wedge \mathtt{out=coff} \wedge \mathtt{wtr'=wtr-1} \vee$$
$$\mathtt{nd=1} \wedge \mathtt{wtr>0} \wedge \mathtt{in=req} \wedge \mathtt{out=tea} \wedge \mathtt{wtr'=wtr-1} \vee$$

6 Experiments

In this section, we present an experimental evaluation of the presented methods. We start by presenting the deployed tool-chain. Thereafter, we present a validation of our method on one case study with another model-based mutation testing tool. Finally, we present quantitative results on a broad range of generic models.

6.1 Toolchain

Figure 2 shows the toolchain that we use to produce test suites for models encoded in the modeling languages Verilog and SMV. Verilog models are deterministic while SMV models can be non-deterministic.

Variable annotation. As a first step, we annotate variables as inputs and outputs. These annotations were added manually for Verilog, and heuristically for SMV (partitioning variables into outputs and inputs).

Mutation and transformation. We produce conditional mutants via a mutation engine. For Verilog, we implemented our own mutation engine into the open source Verilog

compiler VL2MV [12]. We use standard mutation operators, replacing arithmetic operators, Boolean relations, Boolean connectives, constants, and assignment operators. The list of mutation operators used for Verilog can be found in the Appendix of [16]. For SMV models, we use the NuSeen SMV framework [5,6], which includes a mutation engine for SMV models. The mutation operators used by NuSeen are documented in [5]. We implemented the transformation presented in Section 5 into NuSeen and applied it to conditional mutants.
Translation. The resulting conditional mutants from both modeling formalisms are translated into AIGER circuits [9]. AIGER circuits are essentially a compact representation for finite models. The formalism is widely used by model checkers. For the translation of Verilog models, VL2MV and the ABC model checker are used. For the translation of SMV models, NuSMV is used.
Test suite creation. We obtain a test suite, by model checking $\neg\phi_1(\mathcal{I}, \mathcal{O})$ on conditional mutants. Tests are obtained as counter-examples, which are finite prefixes of π-witnesses to $\phi_1(\mathcal{I}, \mathcal{O})$. In case we can not find a counter-example, and use a complete model checking method, the mutant is provably equivalent.

Case study test suite evaluation. We compare the test suite created with our method for a case study, with the model-based mutation testing tool MoMuT [2,15]. The case study is a timed version of a model of a car alarm system (CAS), which was used in the model-based test case generation literature before [4,3,15].

To this end, we created a test suite for a SMV formulation of the model. We evaluated its strength and correctness on an Action System (the native modeling formalism of MoMuT) formulation of the model. MoMuT evaluated our test suite by computing its mutation score — the ratio of killed- to the total number of- mutants— with respect to Action System mutations, which are described in [15].

This procedure evaluates our test suite in two ways. Firstly, it shows that the tests are well formed, since MoMuT does not reject them. Secondly, it shows that the test suite is able to kill mutants of a different modeling formalism than the one it was created from, which suggests that the test suite is also able to detect faults in implementations.

We created a test suite consisting of 61 tests, mapped it to the test format accepted by MoMuT. MoMuT then measured the mutation score of our translated test suite on the Action System model, using Action System mutants. The measured mutation score is 91% on 439 Action System mutants. In comparison, the test suite achieves a mutation score of 61% on 3057 SMV mutants. Further characteristics of the resulting test suite are presented in the following paragraphs.

Quantitative Experiments. All experiments presented in this section were run in parallel on a machine with an Intel(R) Xeon(R) CPU at 2.00GHz, 60 cores, and 252GB RAM. We used 16 Verilog models which are presented in [18], as well as models from opencores.org. Furthermore, we used 76 SMV models that were also used in [5]. Finally, we used the SMV formalism of CAS. All models are available in [1]. Verilog and SMV experiments were run using property driven reachability based model checking with a time limit of 1 hour. Property driven reachability based model checking did not perform well for CAS, for which we therefore switched to bounded model checking with a depth limit of 100.

Characteristics of models. Table 1 present characteristics of the models. For Verilog and SMV, we present average (μ), standard deviation (σ), minimum (Min), and maximum (Max) measures per model of the set of models. For some measurements, we additionally present average (Avg.) or maximum (Max) number over the set of mutants per model. We report the size of the circuits in terms of the number of inputs (#Input), outputs (#Output), state (#State) variables as well as *And* gates (#Gates), which corresponds to the size of the transition relation of the model. Moreover, the row "Avg. Δ # Gates" shows the average size difference (in $\%$ of $\#$ Gates) of the conditional mutant and the original model, where the average is over all mutants. The last row of the table shows the number of the mutants that are generated for the models.

We can observe that our method is able to handle models of respectable size, reaching thousands of gates. Furthermore, $\Delta\#$ Gates of the conditional mutants is relatively low. Conditional mutants allow us to compactly encode the original and mutated model in one model. Hyperproperties enable us to refer to and juxtapose traces from the original and mutated model, respectively. Classical temporal logic does not enable the comparison of different traces. Therefore, mutation analysis by model checking classical temporal logic necessitates strictly separating traces of the original and the mutated model, resulting in a quadratic blowup in the size of the input to the classical model-checker, compared to the size of the input to the hyperproperty model-checker.

Table 1: Characteristics of Models

Parameters	Verilog				SMV				CAS
	μ	σ	Min	Max	μ	σ	Min	Max	
# Models	16				76				1
# Input	186.19	309.59	4	949	8.99	13.42	0	88	58
# Output	176.75	298.94	7	912	4.49	4.26	1	28	7
# State	15.62	15.56	2	40	-	-	-	-	-
# Gates	4206.81	8309.32	98	25193	189.12	209.59	7	1015	1409
Avg. Δ # Gates	3.98%	14.71%	-10.2%	57.55%	8.14%	8.23%	0.22%	35.36%	0.86%
# Mutants	260.38	235.65	43	774	535.32	1042.11	1	6304	3057

Model checking results. Table 2 summarizes the quantitative results of our experiments. The quantitative metrics we use for evaluating our test generation approach are the mutation score (i.e. percentage of killed mutants) and the percentage of equivalent mutants, the number of generated tests, the amount of time required for generating them and the average length of the test cases. Furthermore, we show the number of times the resource limit was reached. For Verilog and SMV this was exclusively the 1 hour timeout. For CAS this was exclusively the depth limit 100.

Finally, we show the total test suite creation time, including times when reaching the resource limit. The reported time assumes sequential test suite creation time. However, since mutants are model checked independently, the process can easily be parallelized, which drastically reduces the total time needed to create a test suite for a model. The times of the Verilog benchmark suite are dominated by two instances of the secure hashing algorithm (SHA), which are inherently hard cases for model checking.

We can see that the test suite creation times are in the realm of a few hours, which collapses to minutes when model checking instances in parallel. However, the timing measures really say more about the underlying model checking methods than our proposed technique of mutation testing via hyperporperties. Furthermore, we want to stress that our method is agnostic to which variant of model checking (e.g. property driven reachability, or bounded model checking) is used. As discussed above, for CAS switching from one method to the other made a big difference.

The mutation scores average is around 60% for all models. It is interesting to notice that the scores of the Verilog and SMV models are similar on average, although we use a different mutation scheme for the types of models. Again, the mutation score says more about the mutation scheme than our proposed technique. Notice that we can only claim to report the mutation score, because, besides CAS, we used a complete model checking method (property driven reachability). That is, in case, for example, 60% of the mutants were killed and no timeouts occurred, then 40% of the mutants are provably equivalent. In contrast, incomplete methods for mutation analysis can only ever report lower bounds of the mutation score. Furthermore, as discussed above, the 61.7% of CAS translate to 91% mutation score on a different set of mutants. This indicates that failure detection capability of the produced test suites is well, which ultimately can only be measured by deploying the test cases on real systems.

Table 2: Experimental Results

Metrics	Verilog				SMV				CAS
	μ	σ	Min	Max	μ	σ	Min	Max	
Mutation Score	56.82%	33.1%	4.7%	99%	64.79%	30.65%	0%	100%	61.7 %
Avg. Test-case Len.	4.26	1.65	2.21	8.05	15.41	58.23	4	461.52	5.92
Max Test-case Len.	21.62	49.93	3	207	187.38	1278.56	4	10006	9
Avg. Runtime	83.08s	267.53s	0.01s	1067.8s	1.2s	5.48s	-	46.8s	7.8s
Equivalent Mutants	33.21%	32.47%	0%	95.3%	35.21%	30.65%	0%	100%	0%
Avg. Runtime	44.77s	119.58s	0s	352.2s	0.7s	2.02s	-	14.9s	-
# Resource Limit	9.96%	27.06%	0%	86.17%	3.8%	19.24%	0%	100%	38.34 %
Total Runtime	68.58h	168.62h	0h	620.18h	0.4h	1.19h	0h	6.79h	1.15h

7 Related Work

A number of test case generation techniques are based on model checking; a survey is provided in [19]. Many of these techniques (such as [30,28,21]) differ in abstraction levels and/or coverage goals from our approach.

Model checking based mutation testing using trap properties is presented in [20]. Trap properties are conditions that, if satisfied, indicate a killed mutant. In contrast, our approach directly targets the input / output behavior of the model and does not require to formulate model specific trap properties.

Mutation based test case generation via module checking is proposed in [10]. The theoretical framework of this work is similar to ours, but builds on module checking instead of hyperproperties. Moreover, no experimental evaluation is given in this work.

The authors of [4] present mutation killing using SMT solving. In this work, the model, as well as killing conditions, are encoded into a SMT formula and solved using specialized algorithms. Similarly, the MuAlloy [31] framework enables model-based mutation testing for Alloy models using SAT solving. In this work, the model, as well as killing conditions, are encoded into a SAT formula and solved using the Alloy framework. In contrast to these approaches, we encode only the killing conditions into a formula. This allows us to directly use model checking techniques, in contrast to SAT or SMT solving. Therefore, our approach is more flexible and more likely to be applicable in other domains. We demonstrate this by producing test cases for models encoded in two different modeling languages.

Symbolic methods for weak mutation coverage are proposed in [8] and [7]. The former work describes the use of dynamic symbolic execution for weakly killing mutants. The latter work describes a sound and incomplete method for detecting equivalent weak mutants. The considered coverage criterion in both works is weak mutation, which, unlike the strong mutation coverage criterion considered in this work, can be encoded as a classic safety property. However, both methods could be used in conjunction with our method. Dynamic symbolic execution could be used to first weakly kill mutants and thereafter strongly kill them via hyperproperty model checking. Equivalent weak mutants can be detected with the methods of [7] to prune the candidate space of potentially strongly killable mutants for hyperpropery model checking.

A unified framework for defining multiple coverage criteria, including weak mutation and hyperproperties such as unique-cause MCDC, is proposed in [24] . While strong mutation is not expressible in this framework, applying hyperproperty model checking to the proposed framework is interesting future work.

8 Conclusion

Our formalization of mutation testing in terms of hyperproperties enables the automated model-based generation of tests using an off-the-shelf model checker. In particular, we study killing of mutants in the presence of non-determinism, where test-case generation is enabled by a transformation that makes non-determinism in models explicit and controllable. We evaluated our approach on publicly available SMV and Verilog models, and will extend our evaluation to more modeling languages and models in future work.

References

1. Mutation testing with hyperproperies benchmark models. `https://git-service.ait.ac.at/sct-dse-public/mutation-testing-with-hyperproperties`. Uploaded: 2019-04-25.
2. B. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. MoMuT::UML model-based mutation testing for UML. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, ICST, pages 1–8, April 2015.
3. Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Softw. Test., Verif. Reliab.*, 25(8):716–748, 2015.
4. Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. 2014.
5. Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Using mutation to assess fault detection capability of model review. *Softw. Test., Verif. Reliab.*, 25(5-7):629–652, 2015.
6. Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Nuseen: A tool framework for the nusmv model checker. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 476–483. IEEE Computer Society, 2017.
7. Sébastien Bardin, Mickaël Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. Sound and quasi-complete detection of infeasible test requirements. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10, 2015.
8. Sébastien Bardin, Nikolai Kosmatov, and François Cheynier. Efficient leveraging of symbolic execution to advanced coverage criteria. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 173–182, 2014.
9. Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond, 2011. Available at `fmv.jku.at/hwmcc11/beyond1.pdf`.
10. Sergiy Boroday, Alexandre Petrenko, and Roland Groz. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science*, 190(2):3–19, 2007.
11. Timothy A Budd, Richard J Lipton, Richard A DeMillo, and Frederick G Sayward. Mutation analysis. Technical report, DTIC Document, 1979.
12. Szu-Tsung Cheng, Gary York, and Robert K Brayton. Vl2mv: A compiler from verilog to blif-mv. *HSIS Distribution*, 1993.
13. Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. *Temporal Logics for Hyperproperties*, pages 265–284. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
14. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
15. Andreas Fellner, Willibald Krenn, Rupert Schlick, Thorsten Tarrach, and Georg Weissenbacher. Model-based, mutation-driven test case generation via heuristic-guided branching search. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Formal Methods and Models for System Design (MEMOCODE)*, pages 56–66. ACM, 2017.
16. Andreas Fellner, Mitra Tabaei Befrouei, and Georg Weissenbacher. Mutation Testing with Hyperproperties. *arXiv e-prints*, page arXiv:1907.07368, Jul 2019.
17. Bernd Finkbeiner, Christopher Hahn, and Tobias Hans. Mghyper: Checking satisfiability of HyperLTL formulas beyond the $\exists^* \forall^*$ fragment. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 11138 of *Lecture Notes in Computer Science*, pages 521–527. Springer, 2018.

18. Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 30–48. Springer, 2015.
19. Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.
20. Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 146–162. Springer-Verlag, 1999.
21. Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341. Springer, 2002.
22. William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Software Eng.*, 8(4):371–379, 1982.
23. Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.
24. Michaël Marcozzi, Mickaël Delahaye, Sébastien Bardin, Nikolai Kosmatov, and Virgile Prevosto. Generic and effective specification of structural test objectives. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 436–441, 2017.
25. McMillan, Kenneth L. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.
26. Greg Nelson. A generalization of dijkstra's calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):517–561, October 1989.
27. A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.
28. Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems (ECBS)*, pages 83–91. IEEE, 2001.
29. Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.
30. Willem Visser, Corina S Psreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.
31. Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Mualloy: a mutation testing framework for alloy. In *International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 29–32. IEEE, 2018.