SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Using Mutation to Assess Fault Detection Capability of Model Review

Paolo Arcaini[1], Angelo Gargantini[1]* and Elvinia Riccobene[2]

[1]*Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy*
[2]*Dipartimento di Informatica, Università degli Studi di Milano, Italy*

SUMMARY

Among validation techniques, *model review* is a static analysis approach that can be performed at the early stages of software development, at the specification level, and aims at determining if a model owns certain quality attributes (like completeness, consistency, and minimality). However, the model review capability to detect behavioral faults has never been measured. In this paper, a methodology and a supporting tool for evaluating the fault detection capability of a NuSMV model advisor is presented, which performs an automatic static model review of NuSMV models. The approach is based on the use of *mutation* in a similar way as in mutation testing: Several mutation operators for NuSMV models are defined and the model advisor is used to detect behavioral faults by statically analyzing mutated specifications. In this way, it's possible to measure the model advisor ability to discover faults. To improve the quality of the analysis, the equivalence between a NuSMV model and any of its mutants must be checked. To perform this task, this paper proposes a technique based on the concept of equivalent Kripke structures, since NuSMV models are Kripke structures. A number of experiments assess the fault detecting capability, precision and accuracy of the proposed approach. Analysis of variance is used to check if the results are statistically significant. Some relationships among mutation operators and model quality attributes are also established. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: mutation analysis; model review; fault detection capability; NuSMV; equivalent mutants

1. INTRODUCTION

There exist countless techniques and methodologies for software verification and validation. They include testing, inspections, model analysis and many other methods for quality assurance. Testing can be classified as a *dynamic* technique since it requires the execution of the program under test, while other techniques, which do not require the execution of the code, can be classified as *static* [1]. These static validation techniques (like syntax checking, type checking, code conventions, etc.) are widespread at code level but not at the specification level, albeit it is undoubted that faults discovered early in the software development can be fixed more easily and with reduced cost. The importance of applying static techniques on models is also remarked by Parnas et al. [2] who state that static *application-independent properties* should be defined and proved before applying other more powerful techniques like model checking and formal verification, and that tools automatically performing such checks can save reviewers considerable time and effort.

Among static validation techniques, *model review*, also known as *model inspection*, can be used at the specification level and aims at determining if a model is of sufficient "quality". Arcaini et al. [3]

*Correspondence to: Angelo Gargantini, Dipartimento di Ingegneria, Università degli Studi di Bergamo, viale Marconi 5, 24044 Dalmine (BG), Italy. E-mail: angelo.gargantini@unibg.it introduced an automated model review technique for NuSMV specifications based on the definition of some static *meta-properties* capturing suitable model quality attributes (like completeness, consistency, minimality).

It would be of great importance to be able to assess the capability of tools for static checking in detecting errors in models, especially actual behavioral faults. Indeed, the starting assumption of this work is that some behavioral faults and some static properties (i.e., meta-properties) are related. For instance, the existence of a condition guarding a branch that is always false (discovered by the failure of a suitable meta-property), likely reveals a behavioral fault that prevents the instructions in the branch to be executed. Moreover, consider a meta-property stating the validity of all the specified properties (e.g., invariants, safety, and liveness). Its violation, i.e., the existence of a property that is false, is likely due to a behavioral fault.

The fault detection capability of static analysis has not been studied with the same extension as for testing, since testing explicitly targets behavioral faults. This paper proposes a way to assess the fault detection capability of static model review by using *mutation*.

Mutation is a well known technique in the context of software code, and program mutation consists in introducing small modifications into program code such that these simple syntactic changes, called *mutations*, represent typical mistakes that programmers often make. These faults are deliberately seeded into the original program in order to obtain a set of faulty programs called *mutants*. Program mutation is almost always used in combination with testing. High quality test suites should be able to distinguish the original program from its mutants, i.e., to detect the seeded faults. The history of mutation testing can be traced back to the 70s [4]. Mutation testing has been applied to many programming languages and for any sort of domain application. More recently, it has been applied to specifications like FSMs [5], Petri nets [6], and Statecharts [7], instead of programs. Likewise, this work operates at the specification level and NuSMV is used as a formal method. NuSMV is endowed with a model advisor and a model checker that allows to automate the proposed method. Moreover, there exists a wide repository of NuSMV specification case-studies available for experiments.

The main novelty of the proposed approach is the use of mutation analysis in order to assess the fault detection capability of static model review. The idea behind is quite simple: Is it possible to use the mutation of NuSMV models to assess the quality of the analysis performed by the model advisor? A static analysis like that performed by the model advisor, to be really useful in practice, should be able to distinguish between correct specifications and faulty ones, in a similar way as tests are able to kill mutants in mutation testing. Note that the model advisor does not target behavioral correctness but it is designed to enforce a set of style and consistency rules with the main goal of increasing model qualities, like completeness, consistency, minimality, maintainability, and readability. This creates a *mismatch* between the faults introduced by mutation, which may change the behavior of the models, and the issues that can be detected by the kind of analysis performed by the model advisor, which targets quality attributes of the models. This mismatch is unavoidable and makes the approach, on one side, more unpredictable and, on the other side, contributes to its originality. Faults in the behaviors should ideally be caught by static analysis too, but this is not guaranteed. Indeed, in testing, there always exists a test (possibly difficult to find) that is able to detect a not equivalent mutant, while some not equivalent mutants may be impossible to catch by the proposed static analysis approach.

A set of mutation operators for NuSMV is introduced, representing possible mistakes designers can make. Each operator produces a set of mutated specifications, which however could be equivalent to, i.e., behave as, the original one. Equivalent mutants pose a challenge to any mutation analysis, since they do not represent actual faults and cannot be detected by observing the behavior of the specification. The problem of equivalent mutants is well-known in mutation testing: an equivalent mutant is a mutant that does not change the semantics of the program; therefore, it is impossible to write a test that captures it. In general, detecting equivalent mutants is an undecidable problem [8] and, because it is difficult to automate, a time-consuming activity [9].

Checking the equivalence of mutants improves the rigor of the analysis. Indeed, the analysis based on mutation operators would be more precise if equivalent mutants are not counted as faults.

A novel technique, which is able to automatically detect mutant equivalence by using the NuSMV model checker itself, has been devised independently of the model advisor for the purpose of this study, and it is presented here. Since NuSMV models are a form of Kripke structure, this technique exploits the notion of equivalence between Kripke structures and consists in building a unique *merged* specification and proving for this model a series of CTL properties. This technique is theoretically proved correct [20], and it is here shown to be usable in practice on real examples.

In order to assess the fault detection capability of static model review, a mutation-based process is presented. It is able to classify every mutant in one of four cases, considering if it has been killed or not, and if it is equivalent or not. The ideal outcome would be that the model advisor kills only non-equivalent mutants and does not kill the equivalent ones. However, this is not the case. So, a series of experiments and analysis of variance are used to assess the quality of the proposed method by measuring its sensitivity, precision, and accuracy. The experiments show some interesting relationships among several factors that influence the fault detection capability of the model advisor.

The paper is organized as follows. Section 2 presents the NuSMV syntax and Section 3 presents the NuSMV model advisor. Section 4 introduces a set of mutation operators for NuSMV, representing possible mistakes designers can make. The problem of equivalent mutants is tackled in Section 5. A novel technique for checking equivalence is described in Section 6. Section 7 presents the process that has been devised to combine the use of mutation and static model review. Experiments are presented in Section 8. Related work follows in Section 9 and Section 10 concludes the paper.

2. NUSMV NOTATION

NuSMV [10, 11] is a well-known tool that performs symbolic model checking. It allows the representation of synchronous and asynchronous finite state systems, and the analysis of specifications expressed in *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL). It exploits the Binary Decision Diagrams (BDD) to represent in a concise way the finite state machine under verification. BDDs can be considered as a compressed representation of sets or relations; the transition relation can be expressed in NuSMV as a Boolean formula in two sets of variables, one relative to the current state and the other relative to the next state. BDD size will be used as measure of model complexity.

A NuSMV specification contains a **VAR** section for variable declarations. A variable type can be Boolean, integer defined over intervals or sets, or an enumeration of symbolic constants. A *state* of the model is an assignment of values to variables.

A NuSMV specification describes the behavior of a Finite State Machine (FSM) in terms of a "possible next state" relation between states that are determined by the values of variables. Transitions between states are determined by the updates of the variables declared in the **ASSIGN** section, that contains the initialization (by the instruction **init**) and the update mechanism (by the instruction **next**) of variables. A **DEFINE** statement can also be used as a macro to syntactically replace an *identifier* with the *expression* it is associated with. There exist the following four ways to explicitly assign values to a variable:

ASSIGN identifier := simple_expression −− *simple assignment*
ASSIGN **init**(identifier) := simple_expression −− *init value*
ASSIGN **next**(identifier) := next_expression −− *next value*
DEFINE identifier := simple_expression −− *macro definition*

where *identifier* is a variable identifier; *simple_expression*s are built only from the values of variables in the current state and they can not have a **next** operation inside; *next_expression* relates current and next state variables to express transitions in the FSM (see the NuSMV User Manual [12] for more details on the assignment syntax and restriction rules for assignments). In both *simple-* and *next*-expressions, a variable's value can be determined either unconditionally or conditionally, depending on the form of the expression. Conditional expressions can be:

1. An **if-then-else** expression

```
MODULE main
VAR
    hour: 0..23;
    hour12: 1..12;
    amPm: {AM, PM};
ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 24;
    hour12 :=
        case
            hour in {0, 12}: 12;
            !(hour in {0, 12}): hour mod 12;
        esac;
    amPm :=
        case
            hour < 12: AM;
            hour >= 11: PM;
        esac;

CTLSPEC NAME pmOK := AG(hour > 11 −> amPm = PM);
```

Model 1. NuSMV model of a clock (only hours)

```
cond1 ? exp1 : exp2
```

which evaluates to *exp1* if the condition *cond1* evaluates to true, and to *exp2* otherwise.

2. A condition **case** expression:

```
case
    left_expression_1 : right_expression_1;
    ...
    left_expression_N : right_expression_n;
esac
```

which returns the value of the first *right_expression_i* such that the corresponding *left_expression_i* condition evaluates to TRUE, and the previous *i-1* left expressions evaluate to FALSE. The type of expressions on the left hand side must be Boolean. An error occurs if all expressions on the left hand side evaluate to FALSE. To avoid these kinds of errors, NuSMV performs a static analysis and, if it believes that in some states no left expression may be true, it forces the user to add a *default case* with *left_expression* equal to TRUE.

NuSMV offers another more declarative way of defining initial states and transition relations. Initial states can be defined by the keyword **INIT** followed by characteristic properties that must be satisfied by the variables values in the initial states. Transition relations can be expressed by constraints, through the keyword **TRANS**, on a set of *current state/next state* pairs. *Invariant conditions* can be expressed by the command **INVAR**.

Temporal properties are specified in the **CTLSPEC** (resp. **LTLSPEC**) section that contains the CTL (resp. LTL) properties to be verified. It is possible to name a property specifying an identifier after the keyword *NAME*. Naming the properties permits to ask the model checker to check only a particular property.

Model 1 shows the NuSMV model of a clock that memorizes only the hours: the variable $hour$ provides the hour in the 24-hour format, whereas variables $hour12$ and $amPm$ provide the hour in the 12-hour format. Variable $hour$ is initialized to zero and it is incremented of a unit (modulo 24) in each transition from a state to the next one. The values of $hour12$ and $amPm$ are defined based on the value of $hour$. A CTL property checks that, in each state, if $hour$ is greater than 11 then $amPm$ is PM; the property identifier is $pmOK$.

3. AUTOMATIC STATIC REVIEW OF NUSMV MODELS

Simpler forms of model static analysis are already guaranteed in NuSMV by the semantics of the language. For instance, consistency of assignments to variables, one of the main goals of other model review techniques [13], is guaranteed by certain restrictive syntactic rules on the structure of assignments [12]. The restriction rules for assignments are: a) *The single assignment rule*: each variable cannot have multiple conflicting definitions; b) *The circular dependency rule*: a set of equations must not have "cycles" in its dependency graph not broken by delays.

Arcaini et al. developed a methodology and a tool for automatic review of NuSMV models [3, 14]. They identified model attributes and characteristics that should hold in any NuSMV model, independently from the particular model to analyze. These quality attributes can be expressed by the following formal predicates, called *meta-properties* (MP).

MP1 *Every assignment condition can be true.*
MP2 *Every assignment is eventually applied.*
MP3 *The assignment conditions are mutually exclusive.*
MP4 *For every assignment terminated by a default condition* true, *at least one assignment condition is true.*
MP5 *No assignment is always trivial.*
MP6 *Every variable can take any value in its domain.*
MP7 *Every variable not explicitly assigned is used.*
MP8 *Every independent variable is used.*
MP9 *Every property is proved true.*
MP10 *No property is vacuously satisfied.*

These MPs should be true in order for a NuSMV model to have the required quality attributes. Therefore, they can be assumed as measures of model quality. MPs refer to the following categories of model quality attributes.

- *Consistency* requires that there are no model statements (variable assignments, propriety specifications, behaviors, etc.) that conflict with each other (e.g., MP3 and MP9).
- *Completeness* requires that every system behavior is explicitly modeled. This encourages the explicit assignment of variables (MP7) and that at least one assignment condition, apart from the default condition, is true (MP4).
- *Minimality* guarantees that the specification does not contain elements – i.e., variables, assignments, domain elements, etc. – defined or declared in the model but never used. Minimality of the assignments requires that every assignment can be performed (MP1, MP2) and it is really useful (MP5). Every value in the domains should be necessary (MP6) and every variable used (MP7 and MP8). Minimality of properties requires that property specifications are not vacuously satisfied (MP10). These defects are also known as "over-specification".

In Model 1, for example, the assignment of variable $amPm$ violates meta-property MP3, since the conditions $hour < 12$ and $hour >= 11$ are not mutually exclusive: both conditions are true when variable $hour$ takes value 11. Note that, however, the definition of variable $amPm$ is correct since, when $hour$ takes value 11, $amPm$ correctly assumes value AM (because the first branch of the case expression whose condition evaluates to true is taken). In this case, the violation of the meta-property has indicated a stylistic defect, not a real fault.

4. MUTATION OPERATORS FOR NUSMV MODELS

In order to generate *mutated specifications* (or *mutants*), some *mutation operator*s must be defined, i.e., rules that specify syntactic variations of the specification [8]. The standard way is to derive the operators directly from fault classes, namely errors that can be introduced by the developer in the specification: typical fault classes are those defined by Kuhn [15].

```
MODULE main
VAR
    hour: 0..23;
    hour12: 1..12;
    amPm: {AM, PM};
ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 24;
    hour12 :=
        case
            hour in {0, 12}: 12;
            !(hour in {0, 12}): hour mod 12;
        esac;
    amPm :=
        case
            hour >= 11: PM;
            hour < 12: AM;
        esac;

CTLSPEC NAME pmOK := AG(hour > 11 -> amPm = PM);
```

Model 2. Mutation of Model 1 – Swapped branches

Some mutation operators have been identified: some of them are the usual ones described in literature (e.g., LOR, SA0), others are more specific operators tailored on NuSMV specifications (e.g., MB, SB). Mutation operators can be classified as follows:

* *Structure mutation operators* modifying the structure of the specification:
    - *Missing Branch* (MB): in a case expression one of the branches is removed.
    - *Swapped Branches* (SB): in a case expression two branches are swapped. Given a case expression with n branches, it produces $n\cdot(n-1)/2$ mutants. Model 2 is a mutant of Model 1, obtained by swapping the two branches of the case expression of the assignment of variable $amPm$.
    - *Missing Definition* (MD): a variable assignment (simple, init or next) is removed from an ASSIGN section.
    - *Missing TRANS/INIT/INVAR* (MTC/MIC/MINC): a TRANS/INIT/INVAR constraint is removed from the specification.
* *Expression mutation operators* modifying expressions:
    - *Expression Negation* (EN): an expression is replaced by its negation.
    - *Logical Operator Replacement* (LOR): a logical operator (&, |, $\rightarrow$, $\leftrightarrow$ xor, $xnor$) is replaced by another logical operator.
    - *Mathematical Operator Replacement* (MOR): a mathematical operator ($+$, $-$, $*$, $/$, mod) is replaced by another one.
    - *Relational Operator Replacement* (ROR): a relational operator ($=$, $!=$, $<$, $<=$, $>$, $>=$) is replaced by another one. Model 3 is a mutant of Model 1, obtained by replacing, in the second condition of the case expression of the assignment of variable $amPm$, the operator $>=$ with the operator $>$.
    - *Stuck-At 0/1* (SA0, SA1): a boolean expression is replaced by the value FALSE/TRUE.
    - *Associative Shift* (AS): in an expression, operators precedence is changed introducing and/or removing parentheses (e.g., $(a \mid b)$ & c is mutated in $a \mid (b$ & $c))$.
* *Value mutation operators* modifying the occurrence of numerical or enumerative values:

```
MODULE main
VAR
    hour: 0..23;
    hour12: 1..12;
    amPm: {AM, PM};
ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 24;
    hour12 :=
        case
            hour in {0, 12}: 12;
            !(hour in {0, 12}): hour mod 12;
        esac;
    amPm :=
        case
            hour < 12: AM;
            hour > 11: PM;
        esac;

CTLSPEC NAME pmOK := AG(hour > 11 -> amPm = PM);
```

Model 3. Mutation of Model 1 – Relational Operator Replacement

- *Enumeration Replacement* (ER): an enumeration constant is replaced by another enumeration constant belonging to the same domain.
- *Number Replacement* (NR): an integer constant n is replaced by the integer constant $n + 1$ or $n - 1$.
- *Digit Replacement* (DR): given an integer constant n whose decimal representation is $n_m \dots n_1$, a digit n_i (with $i = 1, \dots, m$) is replaced with a different digit (e.g., 98 is mutated in 92). For each integer constant, it produces $9 \cdot m$ mutants.

All the mutation operators previously introduced are applied to the ASSIGN, DEFINE, TRANS, INIT and INVAR sections; in this work neither the variable declaration (VAR section) nor the properties specifications (CTLSPEC/LTLSPEC section) are mutated.

Since neither variable domains nor modules instances are modified, all the produced mutant specifications have the same state space of the original specification; the applied mutations can only change the transition relation, the set of reachable states and the set of initial states.

Although the operators introduce only single mutations (first order), they can be applied in sequence in order to obtain higher order mutants [16].

In the rest of the paper mutants will be identified with the acronym of the mutation operator that generates them.

Note that the mutation operators are deliberately not directly related to the meta-properties that are addressed by the NuSMV model advisor. Indeed, mutation operators express faults that the designers can make during the specification activity, while meta-properties refer to quality attributes of the models. As already emphasized before, the model advisor is not designed to target behavioral faults.

5. EQUIVALENT MUTANTS

When a mutant behaves like the original model, it is said to be *equivalent*. Most mutation operators can produce equivalent mutants, which pose a challenge to mutation analysis, since they do not represent actual faults and can not be detected by observing the behavior of the specification. The problem of functionally equivalent mutants is well-known in mutation testing [8, 9]: an equivalent mutant does not change the semantics of the program. Since the semantics of the program is unchanged, it is impossible (and useless) to write a test that captures it.

Model 2 is a non-equivalent mutant of Model 1, since the seeded mutation (the swapping of the case expression branches in the assignment of variable $amPm$) has changed the behavior of the model. In fact, when variable $hour$ takes value 11, the variable $amPm$ takes value AM in Model 1, whereas it takes value PM in Model 2.

Model 3, instead, is an equivalent mutant of Model 1, since the seeded mutation (the replacement of the relational operator $>=$ with operator $>$ in the second condition of the case expression of the assignment of variable $amPm$) has not changed the behavior of the model. In fact, in Model 1 the condition $hour >= 11$ is checked only if $hour$ is greater than 11, since the previous branch is not taken if condition $hour < 12$ is false. So, replacing the second condition with $hour > 11$ does not affect the behavior of the model.

Although detecting equivalent mutants is in general an undecidable problem [8] and, when possible, is a time-consuming activity [9] and difficult to automatize, the following section presents a technique (based on the equivalence [17, 18] of Kripke structures [19]) able to discover NuSMV equivalent mutants.

6. DISCOVERING EQUIVALENT NUSMV SPECIFICATIONS

This section briefly introduces the technique, described in detail in a previous work [20], that permits to check if two NuSMV specifications having the same state space are equivalent. In Section 7.1, this technique is used to check if a mutant is equivalent to the original specification. The reader who is not interested in the mathematical formulation of the equivalence, can skip this section.

6.1. Background on the equivalence for Kripke Structures

In the following, the essential definitions and theorems on Kripke structures [19] and their equivalence [17, 18] are presented. Such results will be exploited in Section 6.2 to detect equivalence between NuSMV specifications, since they are a form of Kripke structures.

Definition 1 (Kripke structure). A *Kripke structure* is a quadruple $M = \langle S, S^0, T, \mathcal{L}\rangle$ where

- S is a set of states;
- $(S^0 \subseteq S) \neq \varnothing$ is the set of initial states;
- $T \subseteq S \times S$ is the transition relation that is left-total, i.e., $\forall s \in S, \exists s' \in S \; ((s, s') \in T)$;
- $\mathcal{L} : S \rightarrow \mathcal{P}(AP)$ is the proposition labeling function, where AP is a set of atomic propositions and $\mathcal{P}(AP)$ its powerset.

Note that Def. 1 includes Kripke structures having different states, but equally labeled. Since these models cannot be represented in NuSMV, it is required $\mathcal{L}$ to be injective (i.e., $\forall s_1, s_2 \in S \; (s_1 \neq s_2 \rightarrow \mathcal{L}(s_1) \neq \mathcal{L}(s_2))$), to guarantee that a state is uniquely identified by its labels.

Definition 2 (Computation tree). Given a Kripke structure $M = (S, S^0, T, \mathcal{L})$, a computation tree of M is a tree structure where the root is an initial state $s_0 \in S^0$, and the children of a node $s \in S$ in the computation tree are all the states $s' \in S$ such that there exists a transition $(s, s') \in T$.

Definition 3 (Structure equivalence). Two Kripke structures M_1 and M_2 with the same set of atomic propositions are equivalent iff they have the same computation trees.

Definition 4 (Next and reachable states). Let $M = \langle S, S^0, T, \mathcal{L}\rangle$ be a Kripke structure. A state s' is *next* of another state s if $(s, s') \in T$. $next(s)$ denotes the set of the next states of s. A state $s \in S$ is reachable in M if there exists a path $s_0, \ldots, s_n$ such that $s_0 \in S^0$, $s_n = s$, and $(s_i, s_{i+1}) \in T$ for all $i \in [0, n-1]$. $reach(M) \subseteq S$ denotes the set of reachable states of the structure M.

Let $M_1 = \langle S_1, S_1^0, T_1, \mathcal{L}_1\rangle$ and $M_2 = \langle S_2, S_2^0, T_2, \mathcal{L}_2\rangle$ be two Kripke structures with the same set of atomic propositions AP. A relation E can be defined on $S_1 \times S_2$ to express the equivalence between states of the two structures M_1 and M_2; two states are equivalent if they have the same labels and bring to next states having the same labels.

Definition 5 (**State equivalence**). $\forall s_1 \in S_1 \forall s_2 \in S_2$, s_1 and s_2 are equivalent, i.e., $s_1 E s_2$, iff the following condition holds:

$$\begin{array}{ll}\mathcal{L}_1(s_1) = \mathcal{L}_2(s_2) & \wedge \\ \forall s'_1 \in next(s_1)\ \exists s'_2 \in next(s_2) : \mathcal{L}_1(s'_1) = \mathcal{L}_2(s'_2) & \wedge \\ \forall s'_2 \in next(s_2)\ \exists s'_1 \in next(s_1) : \mathcal{L}_2(s'_2) = \mathcal{L}_1(s'_1) & \end{array}$$

Theorem 1 (**Structure equivalence**). Let M_1 and M_2 be two Kripke structures with the same set of atomic propositions. If the following properties hold (initial states have same labeling and reachable states are equivalent):

$$\forall s_1^0 \in S_1^0, \exists s_2^0 \in S_2^0 \colon \left[\mathcal{L}_1(s_1^0) = \mathcal{L}_2(s_2^0)\right] \tag{1}$$

$$\forall s_2^0 \in S_2^0, \exists s_1^0 \in S_1^0 \colon \left[\mathcal{L}_2(s_2^0) = \mathcal{L}_1(s_1^0)\right] \tag{2}$$

$$\forall s_1 \in reach(M_1)\ \exists s_2 \in reach(M_2) \colon s_1\ E\ s_2 \tag{3}$$

$$\forall s_2 \in reach(M_2)\ \exists s_1 \in reach(M_1) \colon s_2\ E\ s_1 \tag{4}$$

then M_1 and M_2 are equivalent.

6.2. Equivalence checking for NuSMV models

Since mutation operators do not add or remove variables or domain elements, a NuSMV specification and any of its mutants have the same state labels, i.e., set of atomic propositions. For this reason, Theorem 1 could be used for proving structure equivalence, but its application is difficult, since one should compute the sets of initial and reachable states.

A possible way to check the equivalence of the states is to use the model checker itself over a unique model of the two specifications. The trivial solution is renaming all the variables of one of the two specifications, merging them in a unique model, translating Theorem 1 requirements into suitable CTL properties, and verifying them. This approach does not scale. However, it is possible to statically detect the set of variables whose updating expressions (i.e., expressions used to initialize the variable or to define its value in the next state) are modified by the mutation operators: such variables are called *mutated*. A variant of the *cone of influence* technique [19] can be applied to keep only, besides the mutated variables, those variables influencing them.

This section shows how to build the *merged* model, and then presents the properties that have to be guaranteed in order to have the models equivalence. More details, including theorem proofs, can be found in the technical report [20].

Definition 6 (**NuSMV model as Kripke structure**). A NuSMV model is a Kripke stucture $M = \langle S, S^0, T\rangle$ where each state of S is labeled by a predicate $\bigwedge_{i=1}^{p}(v_i = d_i)$, being $var(M) = \{v_1, \ldots, v_p\}$ a finite fixed set of variables and $\{d_1, \ldots, d_p\}$ their interpretation values over domains $D_1, \ldots, D_p$; the transition relation T expresses the updating of the state variables interpretation by the syntax given in Section 2.

In the sequel, $M_o = \langle S_o, S_o^0, T_o\rangle$ denotes a NuSMV model and $M_m = \langle S_m, S_m^0, T_m\rangle$ one of its mutants.

Definition 7 (**Merged Specification**). Given the NuSMV specifications M_o and M_m, a *merged* specification is the NuSMV model $M_e = \langle S_e, S_e^0, T_e\rangle$ built as follows:

- $var(M_e) = MV \cup MV' \cup DV$, being
  - $MV = \{\tilde{v}_1, \ldots, \tilde{v}_k\}$ the set of all variables of M_o whose updating expression has been mutated in M_m.
  - $MV' = \{\tilde{v}'_1, \ldots, \tilde{v}'_k\}$ a renamed copy of MV. There exists a bijective function $mut : MV \rightarrow MV'$ such that $\forall \tilde{v}_i \in MV(mut(\tilde{v}_i) = \tilde{v}'_i)$.

```
MODULE main
VAR
    hour: 0..23;
    amPm: {AM, PM};
    amPmMut: {AM, PM};
ASSIGN
    init(hour) := 0;
    next(hour) := (hour + 1) mod 24;
    amPm :=
        case
            hour < 12: AM;
            hour >= 11: PM;
        esac;
    amPmMut :=
        case
            hour >= 11: PM;
            hour < 12: AM;
        esac;
```

Model 4. Merged specification of Models 1 and 2

  - $DV = \{v_{k+1}, \ldots, v_n\}$ the set of all non mutated variables of M_o which the value of some mutated variables depends on, i.e., $v \in DV$ iff there exists a variable $\tilde{v} \in MV$ whose value in some state is determined according to the value of v in the current/previous state. It yields $MV \cap DV = \varnothing$.

- The initial assignments of variables in $MV \cup DV$ are those declared in M_o, while variables in MV' have initial assignments as in M_m.
- The transition relations of variables in $MV \cup DV$ are those defined in the machine M_o; the transition relations of variables in MV' are those defined in the machine M_m.

For example, Model 4 is the merged specification to use for checking the equivalence between Models 1 (the original model) and 2 (the mutant). The set of variables of the merged specification is composed as follows: $MV = \{amPm\}$ and $MV' = \{amPmMut\}$ since the mutation affects the definition of variable $amPm$, and $DV = \{hour\}$ because the definition of variable $amPm$ depends on variable $hour$. The transitions relation of variables $amPm$ and $hour$ are those specified in the original model (Model 1), whereas the transition relation of variable $amPmMut$ has been derived from the transition relation of variable $amPm$ in the mutated model (Model 2). Note that variable $hour12$ is not included in the merged specification because its transition relation has not been mutated and it is not involved in the definition of a mutated variable.

6.2.1. Checking for equivalence In order to declare the equivalence between M_o and M_m, a set of properties about the initial states and about the transition relation of the machine M_e must be proved.

Definition 8. Given a merged specification M_e, predicates $Both$ and $Either$ are defined as follows:

$$\begin{aligned} Both(d_1, \ldots, d_n) &\triangleq \bigwedge_{i=1}^{k} (\tilde{v}_i = d_i \wedge \tilde{v}'_i = d_i) \wedge \bigwedge_{j=k+1}^{n} v_j = d_j \\ Either(d_1, \ldots, d_n) &\triangleq \left(\bigwedge_{i=1}^{k} \tilde{v}_i = d_i \vee \bigwedge_{i=1}^{k} \tilde{v}'_i = d_i \right) \wedge \bigwedge_{j=k+1}^{n} v_j = d_j \end{aligned}$$

Given an n-tuple of values $d = (d_1, \ldots, d_n)$, $Both(d)$ is true if both the mutated variables $MV = \{\tilde{v}_1, \ldots, \tilde{v}_k\}$ and their renamed copies $MV' = \{\tilde{v}'_1, \ldots, \tilde{v}'_k\}$ take the same values $d_1, \ldots, d_k$. $Both$ identifies the states of the merged specification in which the variables of the original specification and of the mutated specification are equal.

Given an n-tuple of values $d = (d_1, \ldots, d_n)$, $Either(d)$ is true if either the mutated variables $MV = \{\tilde{v}_1, \ldots, \tilde{v}_k\}$ or their renamed copies $MV' = \{\tilde{v}'_1, \ldots, \tilde{v}'_k\}$ take the values $d_1, \ldots, d_k$. $Either$ identifies the states of the merged specification that have a corresponding state in either the original or the mutated specification.

Lemma 1 (Equivalence of initial states)**.** If the following property holds in M_e:

$$\forall d_{i=1}^{n} \in D_i \quad \left(\begin{array}{c} \exists s_0 \in S_0 \quad [\![Either(d_1, \ldots, d_n)]\!]_{s_0} \rightarrow \\ \exists s'_0 \in S_0 \quad [\![Both(d_1, \ldots, d_n)]\!]_{s'_0} \end{array} \right) \tag{5}$$

then Properties 1 and 2 of Theorem 1 hold.

NuSMV itself can be used to prove Formula 5. However, in order to check in NuSMV that a property φ is true in *at least* an initial state (i.e., $\exists s_0 \in S^0 \ (M, s_0) \models \varphi$), a trick must be applied, since in NuSMV a CTL property φ is true ($M \models \varphi$) iff it is true starting from *each* initial state (i.e., $\forall s_0 \in S^0 \ \ (M, s_0) \models \varphi$). The trick consists in using $\neg\varphi$ instead of φ: if $M \models \neg\varphi$ is false, than there exists an initial state in which φ is true.

So, in order to check Formula 5, $\forall d_{i=1}^{n} \in D_i$, one first needs to check the CTL property $\neg Either(d_1, \ldots, d_n)$ and, if it is false, then (s)he must also check that the CTL property $\neg Both(d_1, \ldots, d_n)$ is false as well. As soon as a tuple of values $(\hat{d}_1, \ldots, \hat{d}_n)$ is found such that $\neg Either(\hat{d}_1, \ldots, \hat{d}_n)$ is false and $\neg Both(\hat{d}_1, \ldots, \hat{d}_n)$ is true, then it's possible to state that Formula 5 is false. Otherwise Formula 5 is true.

Lemma 2 (Equivalence of the transition relations)**.** If the following property holds in M_e:

$$\begin{array}{l} \forall d_{i=1}^{n} \in D_i \\ \forall s \in reach(M) \end{array} \left(\begin{array}{l} \left(\bigwedge_{i=1}^{k} [\![\tilde{v}_i]\!]_s = [\![\tilde{v}'_i]\!]_s \wedge \exists s' \in next(s) \ [\![Either(d_1, \ldots, d_n)]\!]_{s'} \right) \rightarrow \\ (\exists s'' \in next(s) \ [\![Both(d_1, \ldots, d_n)]\!]_{s''}) \end{array} \right) \tag{6}$$

then Properties 3 and 4 of Theorem 1 hold.

Checking Formula 6 is equivalent to verify that CTL property

$$\mathtt{AG} \left(\left(\bigwedge_{i=1}^{k} \tilde{v}_i = \tilde{v}'_i \wedge \mathtt{EX}(Either(d_1, \ldots, d_n)) \right) \rightarrow \mathtt{EX}\left(Both(d_1, \ldots, d_n)\right) \right)$$

holds in the specification M_e, $\forall d_{i=1}^{n} \in D_i$.

The proof of both Lemmas 1 and 2 is provided in the technical report [20]. Thanks to these lemmas, Theorem 1 can be applied to prove the equivalence between M_o and M_m.

Example This example shows the application of Lemmas 1 and 2 to check the equivalence of Models 1 and 2. Some of the following CTL properties must be checked against the merged specification (Model 4) in order to prove the equivalence of the two models in their initial states.

CTLSPEC NAME isNotInitState_1 := !((amPm = AM | amPmMut = AM) & hour = 0)
CTLSPEC NAME notEqInitState_1 := !((amPm = AM & amPmMut = AM) & hour = 0)
CTLSPEC NAME isNotInitState_2 := !((amPm = AM | amPmMut = AM) & hour = 1)
CTLSPEC NAME notEqInitState_2 := !((amPm = AM & amPmMut = AM) & hour = 1)
...
CTLSPEC NAME isNotInitState_25 := !((amPm = PM | amPmMut = PM) & hour = 0)
CTLSPEC NAME notEqInitState_25 := !((amPm = PM & amPmMut = PM) & hour = 0)
...
CTLSPEC NAME isNotInitState_48 := !((amPm = PM | amPmMut = PM) & hour = 23)
CTLSPEC NAME notEqInitState_48 := !((amPm = PM & amPmMut = PM) & hour = 23)

One must check that, if a CTL property *isNotInitState_i* is false, then also the CTL property *notEqInitState_i* is false. In the example, *isNotInitState_1* and *notEqInitState_1* are false (i.e., there is an initial state in which *hour* is 0 and both *amPm* and *amPmMut* are *AM*), and all the properties $isNotInitState\_i$, with $i = 2, \ldots, 48$, are true: so the two models are equivalent in the unique initial state. Totally, 49 properties over 96 had to be checked.

The following CTL properties must be checked against the merged specification in order to prove the equivalence of the transition relations of the two models.

CTLSPEC NAME transRelOk_1 :=
    AG((amPm = amPmMut & **EX**((amPm = AM | amPmMut = AM) & hour = 0)) –>
        EX((amPm = AM & amPmMut = AM) & hour = 0))
...
CTLSPEC NAME transRelOk_12 :=
    AG((amPm = amPmMut & **EX**((amPm = AM | amPmMut = AM) & hour = 11)) –>
        EX((amPm = AM & amPmMut = AM) & hour = 11))
...
CTLSPEC NAME transRelOk_36 :=
    AG((amPm = amPmMut & **EX**((amPm = PM | amPmMut = PM) & hour = 11)) –>
        EX((amPm = PM & amPmMut = PM) & hour = 11))
...
CTLSPEC NAME transRelOk_48 :=
    AG((amPm = amPmMut & **EX**((amPm = PM | amPmMut = PM) & hour = 23)) –>
        EX((amPm = PM & amPmMut = PM) & hour = 23))

One must check that all the CTL properties $transRelOk\_i$, with $i = 1, \ldots, 48$ are true. As soon as a property is found false, the checking can be stopped since it's possible to state that the two models are not equivalent. In the example, only the first 12 properties have been checked, since $transRelOk\_12$ is false. So, the original specification (Model 1) and its mutant (Model 2) are not equivalent. If no violation were found, the two models would be proved equivalent.

7. MUTATION AND MODEL REVIEW

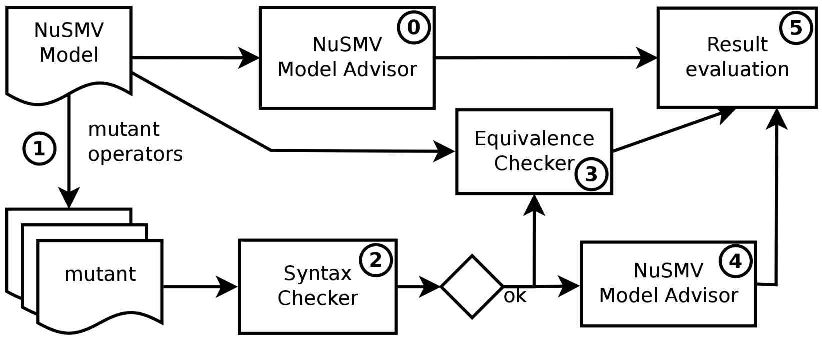

Figure 1. Assessing the fault detection capability of a NuSMV model advisor using mutation analysis

A framework with the aim of assessing the fault detection capability of the NuSMV Model Advisor has been devised. The evaluation process is depicted in Fig. 1, where a NuSMV model is taken as input. The mutation operators presented in Section 4 are applied in order to obtain a large number of mutations of the original NuSMV model (Step 1). Each mutant is then parsed by the original NuSMV parser in order to detect those mutations which result in syntax errors (Step 2). These mutations represent mistakes that can be easily detected by syntax, type, and dependency checking performed by NuSMV itself. *Every* mutant that survives is analyzed by the *equivalence checker* in order to establish if it is equivalent to the original specification (Step 3), and it is checked by the NuSMV model advisor in order to assess the violations of meta-properties (Step 4). The original NuSMV specification has been previously reviewed (Step 0) and the results obtained over the mutant are compared with the original review in order to establish if the mutant is either *killed* or not by the model advisor (Step 5).

Intuitively, a mutant should be killed if the model advisor deems that it has introduced a defect in the original model. The decision of killing a mutant is taken as follows. Let $MpV_i(m)$ be the function that returns the number of violations of meta-property MPi for a NuSMV model m.

Definition 9. The model advisor *kills* a mutant mut if *any* meta-property is violated more times than those of the original specification $orig$, i.e., formally

$$\exists \mathrm{MP}i : MpV_i(mut) > MpV_i(orig)$$

Depending on the number of violations of the original specification, it's possible to identify two scenarios:

1. **Original models without MPs violations**. In case the original model does not violate any meta-property (it represents a completely corrected specification, i.e., $\forall \mathrm{MP}i : MpV_i(orig) = 0$), the mutant is killed if $MpV_i(mut) > 0$ for any MPi.
2. **Original models with MPs violations**: In this case original models violate some meta-properties. This is the most common case, since a model with some violations could still be acceptable (once that the designer has checked that those violations are not faults in the model). This is particularly true for meta-properties that refer to the style in which the specification is written (like MP3 or MP4). In order to kill a mutant, an increase of the number of violations of *any* meta-property suffices. Note that the total number of meta-property violations may decrease due to some mutations: however, in order to analyze the contribution of every single meta-property, it is not suitable considering only the total number of meta-property violations.

7.1. Classification of the mutants

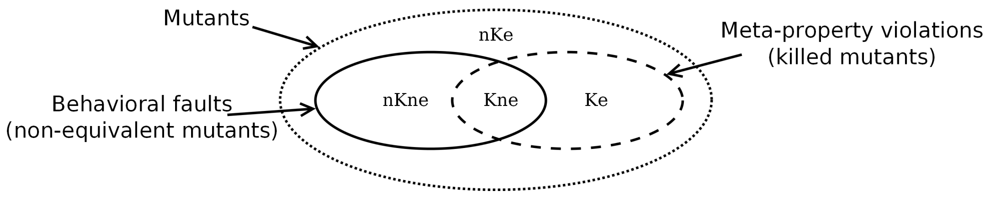

Figure 2. Classification of the mutants

The evaluation of the results takes into account the problem of mutant killing and equivalence. Fig. 2 shows two (possibly overlapping) subsets of the mutants.

Behavioral faults: mutants that are non-equivalent and represent deviations of the correct behavior.

MPs violations: mutants that are killed by the model advisor and represent a decrease of the quality attributes of the model.

The whole set of mutants is therefore partitioned in four subsets, representing the following four cases:

Kne: A non-equivalent mutant (from now on *neq-mutant*) is killed by the model advisor. In this case, the mutation, representing a real fault, causes a violation of some meta-property (or an increase of the number of violated meta-properties). In brief, the model advisor finds a fault that must be fixed.

Ke: An equivalent mutant (from now on *eq-mutant*) is killed by the model advisor. This means that the mutation does not change the behavior of the machine, nevertheless it changes the specification in a way that a meta-property is (more) violated. This represents a *false positive*: the model advisor marks as fault a mutation in the structure which could be classified as "stylistic" defect.

nKe: An eq-mutant is not killed by the model advisor. The mutation does not change the behavior of the machine neither modifies the structure of the specification in a way that some meta-property is violated.

nKne: A neq-mutant is not killed by the model advisor. That means that a real fault passes undetected and represents a *false negative*.

While *Kne* and *nKe* represent the correct outcome of the analysis, *Ke* and *nKne* represent mistakes but with different meanings. False positives (*Ke*) prompt the user to modify a specification that models the correct behavior of the system but that, nevertheless, may have other problems like readability. On the contrary, false negative cases (*nKne*) reveal a weakness of the model review, since it is not powerful enough to discover such faults. Reducing *nKne* would require to introduce new meta-properties (if possible) or performing other kinds of analysis (like simulation and testing).

Fault detection capability If just the fault detection capability is considered, only neq-mutants must be taken into account, since they alone represent real faults. Therefore, the model advisor behaves correctly when it kills neq-mutants (Kne), while it is mistaken when it does not kill neq-mutants (nKne). To increase the fault detection capability, the ratio between Kne and the number of neq-mutants should be increased.

If eq-mutants are also considered, a high number of false positives (Ke) does not diminish the fault detection capability, but it may weaken the quality of the analysis. Indeed, if the number of requested modifications is too high, the designer may decide to ignore the results all together.

8. EXPERIMENTS

The experiments have been executed on a Linux machine, Intel(R) Core(TM) i7 CPU, 4 GB RAM; the model advisor tool is part of the nuseen framework [14]; the NuSMV version used is 2.5.4.

104 NuSMV specifications have been collected from different sources[†]:

- models contained in the NuSMV distribution (9) specifying real systems (including a producer consumer, a bounded retransmission protocol, and a mutual exclusion algorithm);
- specifications sent to the NuSMV mailing list (3);
- specifications found on the Internet (research works, teaching material, etc.) (39);
- specifications internally developed for research and teaching purposes (25);
- NuSMV models obtained from the translation of Abstract State Machine [21] models into NuSMV models through the tool AsmetaSMV [22] (28).

The complexity of the considered models can be measured by the number of BDD variables and the number of BDD nodes allocated. The number of BDD variables is a good indicator of the state space size, since it depends on the number of variables and the size of their domains. The number of BDD nodes allocated reflects the ability of the NuSMV to efficiently store the states symbolically. For the considered models, the average number of BDD variables is 20.44 (that corresponds to a average state space size of 2^{20}), the minimum is 2 and the maximum 156; the average number of BDD allocated nodes is 10650, the minimum is 19 and the maximum 380808.

[†]The used specifications are published at `http://fmse.di.unimi.it/assessingFDC/models.zip`.

Mutation	ER	LOR	MOR	ROR	MB	SB	SA0	SA1
Number of mutants	3037	8540	400	5579	716	1498	3642	3566
Rejected by parser (%)	11	5	38	23	38	0	9	3

Table I. Generated mutants and mutants rejected by the NuSMV parser

The size of a mutant in terms of number of BDD variables is the same as that of its original specification, since variables declarations are not mutated. The number of allocated BDD nodes of a mutant, instead, can be different from that of the original specification because a mutation can change the behavior of a model and so also the structure of the underlying BDDs. However, the number of allocated BDD nodes in mutants is only decreased by 0.18% on average (with a standard deviation of 7.35%).

The experiments have been performed dividing the specifications in the following two sets:

- *NoViolations* containing 51 specifications that do not violate any meta-property;
- *Violations* containing 53 specifications that violate at least one meta-property.

No selection has been made on the specification kind: the sets have been built as much heterogeneous as possible, since the aim is to assess the model advisor fault detection capability not only on a particular kind of specification, but on any possible specification. There are some specifications that are not targeted by the model advisor: indeed, the meta-properties of the model advisor have been originally designed for checking NuSMV specifications that describe the transition relation in an *operational* way (i.e., using the ASSIGN section) and not those that do it in a *declarative* way (i.e., using the INIT and TRANS sections). So, the mutants of specifications belonging to the latter type would probably generate a high number of false negative results (*nKne*). Among all the 104 specifications, 68 are given in an operational style.

The mutated specifications have been obtained applying 8 mutation operators among those described in Section 4: ER, LOR, MOR, ROR, MB, SB, SA0, and SA1. The experiments are based on two classical hypotheses: the *competent programmer* and the *coupling effect*. The proposed approach applies only one mutation operator at the time (first order mutants), even if it supports also higher order mutants (homs) [16], because the number of homs can be very high and because the coupling effect is generally valid for NuSMV specifications as it is for programs [23]. Future work will consider homs in the experiments.

8.1. *Overall results analysis*

Generated mutants 26978 mutants have been generated; 11% (2946) are rejected by the NuSMV parser, whereas the remaining 24032 are syntactically correct NuSMV specifications. Table I reports, for each mutation class, the number of its mutants and the percentage of them rejected by the NuSMV parser.

A lot of MB mutants are rejected by the parser: indeed, most of the mutants in which the default condition of a case expression has been removed are likely detected by the parser since the case conditions become not exhaustive. The SB mutants, instead, can not be detected by the parser: indeed, changing the order of the conditions in a case expression does not modify their exhaustiveness. From now on only mutants that survived the NuSMV parser will be considered.

Execution times The total time taken to execute the NuSMV model advisor over the original specifications is 152 seconds, while executing the NuSMV model advisor over the mutated specifications took about 20 hours. The total time taken to check the equivalence between the original and the mutated specifications is almost 7 hours. Although the process for checking the equivalence is computationally expensive in the worst case (when the mutant is equivalent to the original specification), on average it takes 1.04 seconds for specification, because, when the models are not equivalent, not all the CTL properties introduced in Lemmas 1 and 2 must be checked.

Number of eq-mutants Gruen et al. [9] found that 40% (8/20) of mutants of a Java code were equivalent. Experiments have shown that 27% (6396/24032) of mutants are equivalent. Both results confirm that eq-mutants are a real problem, since their number is not irrelevant.

8.2. Quality Evaluation

This section describes a general evaluation, assessing *sensitivity*, *precision*, and *accuracy* of the proposed approach. According to the ISO standard [24] for binary classification, the quality factors of the proposed methodology can be established following the schema shown in Table II.

Analysis outcome	Mutant		
	non-equivalent	equivalent	
Positive (killed)	true positive *Kne*	false positive *Ke*	→ **Precision**
Negative	false negative *nKne*	true negative *nKe*	
	↓ **Sensitivity**		↘ **Accuracy**

Table II. Quality factors

Note that *sensitivity* considers only neq-mutants, and it measures the actual fault detection capability (since only neq-mutants represent actual faults), whereas *accuracy* measures also the capability of avoiding killing eq-mutants. Finally, *precision* takes into account only mutants that are killed and measures the likelihood that a killed mutant is not equivalent.

In the following, a list of interesting results of experimentation is reported. These results are obtained by formulating some statistical *hypotheses*, guided by specific research questions (**RQ**), and tested using the Kruskal-Wallis one-way analysis of variance [25] that confirmed the hypotheses done by rejecting the corresponding null hypotheses. The Kruskal-Wallis test is used because the data are not normally distributed.

8.2.1. ***Sensitivity*** In order to discover how good the model advisor is in killing neq-mutants, following the terminology of the theory of classification [24], *sensitivity* is introduced as

$$\text{sensitivity} = \frac{\text{Kne}}{\#\textit{neq-mutants}} = \frac{\text{Kne}}{\text{Kne} + \text{nKne}}$$

The overall sensitivity – the actual fault detection capability – of the model advisor in 55%.

RQ1 *Does the sensitivity depend on specifications? If yes, which specification aspects influence it?*

The experiment checks if some of the characteristics of the original specification influence the killability of its neq-mutants. Results show that the number of killed neq-mutants strongly depends on the specification under analysis. Fig. 3 shows the minimum percentage of killed neq-mutants: for example, for the 74% of specifications, at least the 60% of their neq-mutants are killed. Note that, for 28% of the specifications, 100% of neq-mutants are killed. Is it possible to identify any characteristic of the specifications that influences the sensitivity?

Results also show that specifications with meta-property violations (set *Violations*) are more sensitive to mutation, that is the model advisor is more sensitive for specifications with some defects. The model advisor kills more easily neq-mutants of specifications in *Violations* than those in *NoViolations*. On average, 60% of neq-mutants of specifications in *Violations* are killed, whereas the 49% of neq-mutants of specifications in *NoViolations* are killed.

Furthermore, it's possible to note that specifications with temporal properties are more sensitive to mutation. Indeed, there is a correlation between the fact that a specification has temporal properties and the killability of its neq-mutants; indeed, mutants changing the specification behavior can be easily killed by meta-properties MP9 and MP10 checking that each temporal property is,

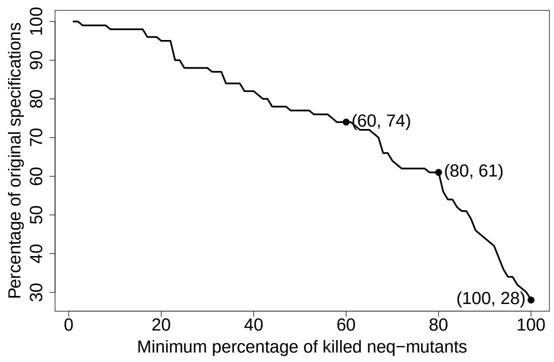

Figure 3. Percentage of at least killed neq-mutants vs percentage of specs

Meta properties – MP	1	2	3	4	5	6	7	8	9	10
Sensitivity (%)	19	24	19	10	5	23	0	0	26	7
Precision (%)	87	85	66	66	91	89	100	100	100	100
Accuracy (%)	39	41	33	30	30	42	27	27	45	31
nKe/eq-mutants (%)	92	88	74	86	99	93	100	100	100	100

Table III. Dependence between the Meta-properties and the Sensitivity, Precision, Accuracy, and Not killed eq-mutants

respectively, true and not vacuously true[‡]. Results show that, on average, 63% of neq-mutants of specifications that have a temporal property are killed, whereas the 41% of neq-mutants of specifications that does not have a temporal property are killed. This result suggests that is a good practice to add temporal properties that redundantly specify what is the desired behavior of the specification.

Finally, a further observation is that specifications in an operational style are more sensitive to mutation. Neq-mutants of specifications whose behavior is given using ASSIGN constructs, are detected more easily than the average. The sensitivity for these mutants raises from 55% to 77%.

RQ2 *Does the sensitivity depend on the meta-properties?*

The experiment wants to discover how much a meta-property is able to kill any kind of neq-mutant. The statistical test confirms that the sensitivity depends on the meta-properties, i.e., there exists a relation between the number of killed neq-mutants and the meta-property used to kill them. Table III (row Sensitivity) reports the percentage of neq-mutants that each meta-property kills. Some meta-properties, like MP7 and MP8, can not detect any mutation; their aim is to identify models where some *monitored* variables (i.e., variables whose value is unbounded) or *independent* variables (i.e., variables that do not depend on any other variable) are never read: it is difficult to obtain such kind of models with the mutation operators that have been tested. MP9 has the greatest capability to kill neq-mutants: a further reason to add desired temporal properties to specifications. Each single meta-property never kills more than 26% of neq-mutants. The model advisor can globally kill more than the half of neq-mutants (55%), which is less than the total sum of all the meta-property sensitivities. This means that some mutants are killed by more than one meta-property and that the meta-property targets are not disjoint.

RQ3 *Is there a relation between neq-mutants killability and their mutation classes?*

[‡]A property is *vacuously* satisfied if that property is satisfied and proved true regardless of whether the model really fulfills what the specifier originally had in mind or not. For example, the LTL property $G(x \rightarrow X(y))$ is vacuously satisfied by any model where *x* is never true. Vacuity is an indication of a problem in either the model or the property.

Mutation	ER	LOR	MOR	ROR	MB	SB	SA0	SA1
Sensitivity (%)	79	51	83	36	81	39	82	40
Precision (%)	82	74	90	75	79	76	79	69
Accuracy (%)	68	51	84	44	71	61	70	44

Table IV. Dependence between the Sensitivity, Precision and Accuracy, and the mutation classes used to generate the mutants

The experiment checks if the neq-mutants of some mutation classes are more easily killed by the model advisor. The statistical test confirms that the neq-mutants killability depends on the class they belong to.

Table IV (row Sensitivity) reports, for each mutation class, the percentage of its neq-mutants that are killed. The model advisor kills more easily some mutants (like MOR and SA0) because their mutation classes often introduce faults in elements of the specification that it checks. There are other mutants, instead, that the model advisor kills very rarely because it is not able to capture most of the faults introduced by their mutation classes. It is interesting that killing SA0 mutants is twice easier than killing SA1 mutants. SA0 and SA1 replace the operands in logical expressions with, respectively, FALSE and TRUE. Such fact can be explained analyzing the specifications and counting the occurrences of logical operators. The & operator occurs 1446 times, whereas the | operator occurs 223 times.

Operator	&	|	$\rightarrow$	xor	$\leftrightarrow$	$xnor$
# of occurrences	1446	223	19	20	0	0

If one sets to FALSE an operand of an AND-expression, the expression is forced to be false in all the states; if in the original specification the expression is true in some state (this is highly probable), it is highly probable that the behavior of the specification is changed and that the model advisor can kill the mutant.
If one sets to TRUE an operand of an AND-expression, instead, the expression is not forced to assume a truth value in all the states: the value of the other operand is still necessary to evaluate the overall expression. So, for the model advisor it is more difficult to kill the mutant, since the mutation could have not changed the behavior of the specification.

For OR-expressions the reasoning is the opposite: the model advisor kills more easily SA1 mutants rather than SA0 mutants. But, since the number of AND-expressions is more than six times the number of OR-expressions (and so the number of their mutants), the number of killed SA0 mutants is higher than the number of killed SA1 mutants.

8.2.2. ***Precision*** Following the terminology of the theory of classification [24], *precision* is defined as

$$\text{precision} = \frac{\text{Kne}}{\#\textit{killed-mutants}} = \frac{\text{Kne}}{\text{Kne} + \text{Ke}}$$

Precision indicates what is the probability that, once the model advisor kills a mutant, the mutant is not equivalent. The overall precision of the model advisor is 76%.

RQ4 *Does the precision depend on the meta-properties?*

The experiment checks what is the probability that a mutant killed with a given meta-property is not equivalent. The statistical test confirms that the precision of the technique depends on the meta-properties used. Table III (row Precision) reports the relation between the precision and the meta-property used. Eight meta-properties are quite precise: when they kill a mutant, it is highly probable that it is not equivalent. Only MP3 and MP4 are not so precise, since they can kill also stylistic defects that do not change the behavior of the specification; this fact is confirmed by the

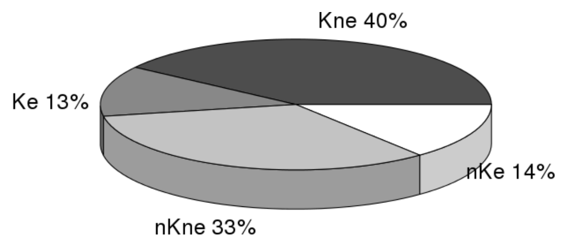

Figure 4. Overall results

ratio between nKe and eq-mutants reported in Table III (row nKe/eq-mutants) where MP3 and MP4 have the lowest ratios in not killing eq-mutants.

RQ5 *Is there a relation between the precision and the mutation classes used to generate the mutants?*

This experiment wants to discover if mutants of some mutation classes permit to obtain better precision in the model advisor. The statistical test confirms that the mutation class used to generate the mutants influences the precision, i.e., faults produced by some mutation classes are captured by the model advisor with a greater precision.

Table IV (row Precision) reports, for each mutation class, the precision of the model advisor over the mutants of that class. Results presented above show that it is difficult to kill the neq-mutants of ROR (Table IV - Sensitivity 36%). Nevertheless, when a mutant of ROR is killed, it is not equivalent with 75% of probability (Table IV - Precision).

8.2.3. ***Accuracy*** The aim of this experiment is to perform a more general evaluation; it does not only consider how good is the model advisor in killing neq-mutants, but also how often the model advisor does not kill the equivalent ones.

Indeed, as seen previously, the model advisor can also kill eq-mutants since some meta-properties can identify stylistic defects that do not change the behavior of the specification.

Following the terminology of the theory of classification [24], *accuracy* is defined as

$$\text{accuracy} = \frac{\text{Kne} + \text{nKe}}{\#mutants} = \frac{\text{Kne} + \text{nKe}}{\text{Kne} + \text{Ke} + \text{nKne} + \text{nKe}}$$

Fig. 4 shows the classification of all the mutants. The value of the overall accuracy of the model advisor is 54% (Kne % + nKe %).

RQ6 *Does accuracy depend on the meta-properties?*

The aim of this experiment is to discover how much a meta-property is able to kill any kind of neq-mutant and not kill the equivalent ones. The statistical test confirms that also the accuracy depends on the meta-property used.

Table III (row Accuracy) reports the accuracy of each meta-property. The accuracy of each meta-property is always greater than its sensitivity: this means that each meta-property is good in not killing eq-mutants. This fact is confirmed by Table III (row nKe/eq-mutants). In general, each meta-property kills very few eq-mutants. However, Table VI shows that 47% of the eq-mutants are killed: this means that the abilities of not killing eq-mutants of the meta-properties are partially disjoint.

RQ7 *Is there a relation between the accuracy and the mutation classes used to generate the mutants?*

The experiment wants to discover if the mutants of some mutation classes permit to obtain better accuracy in the model advisor. The statistical test confirms that the accuracy depends also on the mutation classes, i.e., some mutation classes generate mutants that are captured by the model advisor with a greater accuracy.

Mutation	MP	Sensitivity (%)	Precision (%)	Accuracy (%)
MOR	MP6	48.68	100	68.42
MOR	MP5	38.16	100	61.94
SA0	MP6	53.12	93.88	61.87
SB	MP6	28.57	87.16	60.41
SB	MP3	31.12	81.06	60.15
MOR	MP2	34.21	100	59.51
SB	MP2	33.29	73.94	58.95
MOR	MP1	32.89	100	58.7
SB	MP1	17.22	87.1	55.34
SB	MP9	18.75	77.37	54.61
...			...	
ROR	MP8	0	100	26.18
SA1	MP4	3.02	35.21	26.08
LOR	MP7	0	100	25.85
LOR	MP8	0	100	25.85
SA0	MP7	0.48	100	24.62
SA0	MP8	0.16	100	24.38
ER	MP3	16.43	60.33	22.17
SA0	MP3	2.55	18.82	17.87
ER	MP7	0	100	17.53
ER	MP8	0	100	17.53

Table V. Relationship between mutation classes and meta-properties in terms of Accuracy

Table IV (row Accuracy) reports the accuracy of the model advisor over the mutants of each mutation class. The model advisor obtains greater accuracy than sensitivity when checking the mutants of some mutation classes (e.g., SB), and greater sensitivity than accuracy when checking the mutants of other mutation classes (e.g., SA0). For instance, the accuracy with SB mutants (61%) is definitely greater than the sensitivity (39%) since the model advisor very rarely kills SB eq-mutants. Indeed, eq-mutants swapping two case branches could be killed only by meta-properties MP2 and MP4.

8.3. *Further Evaluation*

RQ8 *Is there a relation between mutation classes and meta-properties?*

This experiment checks the correlation between mutation classes and meta-properties in killing mutants, with the aim of discovering which mutations are targeted with the highest/lowest probability by which meta-properties. The experiment evaluates the relationship between a mutation class *MUT* and a meta-property *MP*i by considering the value of the accuracy obtained by using only *MP*i over the mutants produced with *MUT*. The accuracy is the best indicator of the reliability of the proposed approach, since, unlike sensitivity, it also takes into account the not killed eq-mutants. Table V reports the couples (mutation class, meta-property) sorted in descending order by the value of the accuracy: it reports the first and the last ten couples with their values of sensitivity, precision and accuracy.

As expected, the best/worst couples, in general, are composed by those mutation classes and meta-properties that obtain the best/worst results in Tables IV and III (Row Accuracy), that show, respectively, the accuracy of the model advisor in capturing the mutants of each mutation class, and the accuracy of each meta-property in capturing all the mutants.

However, there are some exceptions due to the strong correlation that may (not) exist between a mutation class and a meta-property.

	eq-mutants	neq-mutants
killed	47%	55%
not killed	53%	45%

Table VI. (Not) Equivalent (not) killed mutants

For example, MP5 does not have a good accuracy over all the mutants (30% in Table III); however, considering only the mutants obtained with the mutation class MOR, the accuracy is 61.94%, the second best result in Table V.

Mutants obtained with the mutation class SA0 are captured by the model advisor with a very good accuracy (the second best value in Table IV), and meta-property MP3 has a decent accuracy over all the mutants (fifth best value in Table III). However, MP3 has a very bad accuracy over the mutants of SA0 (third last value in Table V).

RQ9 *Does equivalence influence killability?*

Table VI reports the percentage of killed eq-mutants (first column) and killed neq-mutants (second column). Results show that the model advisor kills more easily neq-mutants than eq-mutants. This suggests that the introduction of behavioral faults (represented by neq-mutants) is likely to reduce the specification qualities (represented by meta-property violations), while eq-mutants likely pass undetected by the model advisor. This confirms the intuition that eq-mutants are more difficult to kill than neq-mutants.

8.4. Overall Evaluation

Besides sensitivity, precision, and accuracy, considering also the ability of killing stylistic defects, the model advisor behaves correctly in the 67% of the cases (*Kne* + *nKe* + *Ke*) as shown in Fig. 4.

In conclusion, based on the statistical analysis performed, a user can assess the expected success of finding faults by the use of the model advisor, depending on the characteristics of the specification, on the type of targeted faults, and on the type of MP violations. Future work could be the definition of an heuristic procedure that, given a specification (with some features) and the list of violated meta-properties, computes the probability that the specification contains each kind of identified fault.

Moreover, note that considering also the mutants rejected by the NuSMV parser as killed, would improve the quality of the analysis, since it would increase only *Kne*. However, this work focuses only on the model advisor.

About the general validity of the results, a threat to validity could be the selection of the set of specifications. To mitigate this, the specifications have been collected from several heterogeneous sources and with different characteristics.

9. RELATED WORK

Mutation testing applied to programs dates back to the 1970's, when the first attempts applied seeded faults to assess the quality of test suites [26, 27]. A comprehensive good survey about mutation testing has been done by Jia and Harman [4]. Mutation analysis has been applied at design level to models too and it is referred as "specification mutation".

An approach similar to mutation testing can be applied to executable specifications. For instance, Woodward introduces mutation operators for algebraic specifications [28], which can be executed via the process of term rewriting. User-guided generated sets of test expressions from a specification are evaluated against these mutants in order to assess the quality of the test expressions.

Other works focus more on defining meaningful mutation operators for formal notations. Liu and Miao [29] introduce mutation operators for Object-Z specifications; they devise five categories of mutation operators: the *Traditional mutations operators* (e.g., *Operator Replacement* (*OR*)), operators for *Z-specific features* (e.g., *Omit State Invariant* (*OSI*)), and other three categories of operators that are related to the object oriented features of Object-Z, *Visibility*, *Inheritance* and *Polymorphism*. The first category is like categories *Expression mutation operators* and *Value mutation operators* defined in Section 4, namely operators that are commonly used in all programming and specification languages; the last four categories, instead, include operators that try to capture some features that are typical of Z/Object-Z, like done in this work for NuSMV with operators in *Structure mutation operators* (see Section 4). Other similar works have been done for Petri Nets, Finite State Machines, Statecharts, and *Estelle* specifications [6, 5, 7, 30].

Some works combine mutation and model-based test generation. Ammann et al. [31] use mutants of NuSMV models to generate, by model checking, test sequences able to detect the seeded faults. Black et al. [32] identify mutation operators for specifications in NuSMV models, still for test generation purposes; such operators correspond to some of the *Expression mutation operators* defined in Section 4 (e.g., their LRO and RRO are like LOR and ROR).

The combination between mutation and model checking for static analysis is used by Lee and Hsiung [33]. In order to estimate the completeness of a set of specification properties P, the (state-graph) model is mutated and it is checked if P can distinguish the mutation. The approach is similar to this work if one considers only MP9 (all the properties are true) and MP10 (all the properties are non-vacuously satisfied). However, the work of Lee and Hsiung tries to assess the completeness of specification properties, while this work tries to assess the capability of *meta*-properties to detect faults.

Another approach that tries to combine static analysis and mutation is presented by Di Guglielmo et al. [34]. They use mutation analysis in combination with a mix between dynamic and static techniques in order to qualify the properties concerning three aspects: vacuity, completeness, and over-specification. Their approach is similar to this work if one considers only a subset of meta-properties but again, they operate at the level of specification properties while this work considers meta-properties.

Up to now mutation analysis has never been applied to assess the quality of model review as well as of any other kind of static analysis technique. The approach presented in this paper differs from mutation testing in several ways. First, it permits to discover equivalent mutants using the NuSMV model checker itself, while, in code mutation, equivalent mutants are very difficult to detect. Second, not all the mutants that are killed represent real faults, since the model advisor does not consider the functional behavior of the model. Third, actual faults are not detected as far as they are in a model without MP violations, while in code testing there always exists a test case that detects any real fault, as difficult it may be to find.

Detecting equivalent mutants in code testing is an undecidable problem [8]; however different techniques have been proposed to detect the non-equivalence between a program and one of its mutants.

Schuler and Zeller [35] propose a technique (integrated in the Javalanche mutation framework [36]) for detecting non-equivalent mutants, that looks for changes in the code coverage: the idea is that *if a mutant changes the coverage of a run, it is more likely to be not equivalent*. In particular the authors define some *impact metrics*, and consider a mutant non-equivalent if its impact is greater than a given threshold.

Nica and Wotawa [37] represent the original program and one of its mutants as a set of constraints: their approach tries to find a *distinguishing* test that shows the non-equivalence between the original program and the mutant. Another approach based on constraints is described by Offutt and Pan [38]. They identify three conditions that, given the execution of a test t, a mutant must fulfil in order to be considered not equivalent: the mutation (mutated statement) must be reachable by t, the state after the execution of the mutated statement must be different from the corresponding state in the execution of the original program, and the final states of the test execution over the original program and the mutated program must be different.

Baldwin and Sayward [39] propose to use compiler optimizations for detecting equivalent mutants; the approach has been extended and implemented by Offutt [40]. The idea is that a mutant can be seen as an optimization or a de-optimization of the original program and so compiler optimization rules can be used to detect equivalent mutants.

Hierons et al. [41] use program slicing to help the user in the identification of equivalent mutants. The authors propose a technique in which a mutated program in simplified in a smaller program (*slice*) that contains only the instructions involved in the mutation; for the developer, analysing this slice is definitely easier that analysing the whole mutated program.

Fraser and Wotawa [42] propose an approach similar to the approach presented in this paper for checking the equivalence of NuSMV models, in the context of test case generation using model checkers. However that approach has some limitations due to the fact that the checked properties are built using the expressions appearing as guards in the model assignments. First, only models containing assignment conditions referring to the current state can be checked; however, NuSMV also permits to write conditions referring to the *next* value of a variable. Solutions for handling this problem are proposed but they would require the introduction of extra variables (called *shadow*). Second, the approach is applicable only to models written in an operational way, i.e., in which the next values of variables are determined through case expressions. The approach presented in this paper does not have these limitations, since the properties used for checking the equivalence are not built upon the expressions used in the model, but upon the possible states of the model: so, any kind of NuSMV model can be checked.

10. CONCLUSION AND FUTURE WORK

This paper proposes a framework to evaluate the fault detection capability of a model advisor for NuSMV specifications. The approach combines mutation with static review, and it consists in seeding faults into NuSMV models and checking whether model review can detect them. This is similar to mutation testing. However, since it is possible to prove equivalence between a model and its mutants thanks to a novel model checking based technique, the assessment of the quality of the analysis can take into account also equivalent mutants. Experiments proved that the model advisor has a satisfactory sensitivity and accuracy, and an excellent precision. Future work includes (a) experimenting the methodology using higher order mutants and (b) defining an heuristic procedure that provides the user with an estimation of the kind of faults occurring in the specification, and hints on how to correct them.

References

1. Sommerville I. *Software Engineering*. 9th edn., Addison Wesley, 2010.
2. Parnas DL. Some Theorems We Should Prove. *HUG '93: 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Springer-Verlag: London, UK, 1994; 155–162.
3. Arcaini P, Gargantini A, Riccobene E. A model advisor for NuSMV specifications. *Innovations in Systems and Software Engineering* 2011; **7**:97–107. URL `http://dx.doi.org/10.1007/s11334-011-0147-2`.
4. Jia Y, Harman M. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 2011; **37**:649–678, doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62.
5. Fabbri SCPF, Delamaro ME, Maldonado JC, Masiero PC. Mutation analysis testing for finite state machines. *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, 1994; 220 –229, doi: 10.1109/ISSRE.1994.341378.
6. Fabbri SCPF, Maldonado JC, Masiero PC, Delamaro ME, Wong E. Mutation Testing Applied to Validate Specifications Based on Petri Nets. *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*, Chapman & Hall, Ltd.: London, UK, 1996; 329–337. URL `http://dl.acm.org/citation.cfm?id=646214.681539`.
7. Fabbri SCPF, Maldonado JC, Sugeta T, Masiero PC. Mutation testing applied to validate specifications based on statecharts. *Proceedings 10th Int Software Reliability Engineering Symp*, 1999; 210–219, doi:10.1109/ISSRE.1999.809326.
8. Ammann P, Offutt J. *Introduction to Software Testing*. 1 edn., Cambridge University Press: New York, NY, USA, 2008.
9. Gruen BJ, Schuler D, Zeller A. The Impact of Equivalent Mutants. *Mutation '09: Proceedings of the 3rd International Workshop on Mutation Analysis*, 2009; 192–199.
10. Cimatti A, Clarke EM, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. *Proceedings International Conference on Computer-Aided Verification (CAV 2002)*, *LNCS*, vol. 2404, Springer, 2002.

11. The NuSMV website. `http://nusmv.fbk.eu/` 2012.
12. Cavada R, Cimatti A, Jochim CA, Keighren G, Olivetti E, Pistore M, Roveri M, Tchaltsev A. NuSMV 2.5 User Manual. `http://nusmv.fbk.eu/` 2010.
13. Heitmeyer C, Jeffords RD, Labaw BG. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(3):231–261.
14. Arcaini P, Gargantini A, Vavassori P. Nuseen: an eclipse-based environment for the nusmv model checker. *Eclipse-IT 2013 - VIII Workshop Italian Eclipse Community*, 2013. URL `http://arxiv.org/abs/1310.2464`.
15. Kuhn DR. Fault Classes and Error Detection Capability of Specification-Based Testing. *ACM Transactions on Software Engineering and Methodology* 1999; **8**(4):411–424.
16. Jia Y, Harman M. Higher order mutation testing. *Information and Software Technology* 2009; **51**(10):1379 – 1393, doi:10.1016/j.infsof.2009.04.016.
17. Browne MC, Clarke EM, Grümberg O. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science* 1988; **59**(1-2):115 – 131, doi:DOI:10.1016/0304-3975(88)90098-9. URL `http://www.sciencedirect.com/science/article/pii/0304397588900989`.
18. Aziz A, Singhal V, Balarin F. Equivalences for fair Kripke structures. *International Colloquium on Automata, Languages and Programming*, Springer-Verlag, 1994; 364–375.
19. Clarke EM, Grümberg O, Peled D. *Model checking*. MIT Press, 2001.
20. Arcaini P, Gargantini A, Riccobene E. Checking equivalence of NuSMV specifications - http://fmlab.di.unimi.it/assessingFDC/eqCheckNuSMVspecs.pdf. *TR 134*, DTI Department, Università degli Studi di Milano 2011. URL `http://fmse.di.unimi.it/assessingFDC/eqCheckNuSMVspecs.pdf`.
21. Börger E, Stärk R. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
22. Arcaini P, Gargantini A, Riccobene E. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, *Lecture Notes in Computer Science*, vol. 5977, Frappier M, Glässer U, Khurshid S, Laleau R, Reeves S (eds.), Springer, 2010; 61–74.
23. Offutt AJ. The coupling effect: fact or fiction. *SIGSOFT Software Engineering Notes* November 1989; **14**:131–140, doi:http://doi.acm.org/10.1145/75309.75324. URL `http://doi.acm.org/10.1145/75309.75324`.
24. JCGM 200:2008 International vocabulary of metrology. Basic and general concepts and associated terms (VIM).
25. Kruskal WH, Wallis WA. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association* 1952; :583–621.
26. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection help for the practicing programmer. *IEEE Computer* Apr 1978; **11**(4):34–41.
27. Hamlet RG. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering* Jul 1977; **3**(4):279–290.
28. Woodward MR. Errors in algebraic specifications and an experimental mutation testing tool. *Software Engineering Journal* 1993; **8**(4):211–224.
29. Liu L, Miao H. Mutation operators for Object-Z specification. *Proceedings 10th IEEE Int. Conf. Engineering of Complex Computer Systems ICECCS 2005*, 2005; 498–506, doi:10.1109/ICECCS.2005.65.
30. De Souza SDRS, Maldonado JC, Fabbri SCPF, De Souza WL. Mutation Testing Applied to Estelle Specifications. *Software Quality Control* December 1999; **8**:285–301, doi:10.1023/A:1008978021407. URL `http://dl.acm.org/citation.cfm?id=599121.599189`.
31. Ammann PE, Black PE, Majurski W. Using model checking to generate tests from specifications. *Proceedings of the Second International Conference on Formal Engineering Methods, 1998*, ICFEM '98, IEEE Computer Society: Washington, DC, USA, 1998; 46 –54.
32. Black PE, Okun V, Yesha Y. Mutation operators for specifications. *Proceedings Fifteenth IEEE International Conference Automated Software Engineering ASE 2000*, 2000; 81–88, doi:10.1109/ASE.2000.873653.
33. Lee TC, Hsiung PA. Mutation Coverage Estimation for Model Checking. *Automated Technology for Verification and Analysis*, *Lecture Notes in Computer Science*, vol. 3299, Wang F (ed.). Springer Berlin / Heidelberg, 2004; 354–368. URL `http://dx.doi.org/10.1007/978-3-540-30476-0_29`.
34. Di Guglielmo L, Fummi F, Pravadelli G. The role of mutation analysis for property qualification. *Formal Methods and Models for Co-Design, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, 2009; 28 –35, doi:10.1109/MEMCOD.2009.5185375.
35. Schuler D, Zeller A. (Un-)covering equivalent mutants. *Third International Conference on Software Testing, Verification and Validation (ICST), 2010*, 2010; 45–54, doi:10.1109/ICST.2010.30.
36. Schuler D, Zeller A. Javalanche: Efficient mutation testing for Java. *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2009; 297–298.
37. Nica S, Wotawa F. Using constraints for equivalent mutant detection. *Proceedings 2nd Workshop on Formal Methods in the Development of Software*, *EPTCS*, vol. 86, Andrés C, Llana L (eds.), 2012; 1–8.
38. Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability* 1997; **7**(3):165–192.
39. Baldwin D, Sayward F. Heuristics for determining equivalence of program mutations. *Technical Report*, DTIC Document 1979.
40. Offutt AJ, Craft WM. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability* 1994; **4**(3):131–154.
41. Hierons R, Harman M, Danicic S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability* 1999; **9**(4):233–262.
42. Fraser G, Wotawa F. Mutant minimization for model-checker based test-case generation. *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007*, IEEE, 2007; 161–168.