This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Fevzi Belli, Christof J. Budnik, Axel Hollmann, Tugkan Tuglular, W. Eric Wong, Model-Based mutation testing - Approach and case studies, Sci. Comput. Program. 120, pp. 25-48

Model-Based Mutation Testing–Approach and Case Studies*

Fevzi Belli[1] Christof J. Budnik[2] Axel Hollmann[3] Tugkan Tuglular[4] W. Eric Wong[5]

[1]Department of Computer Science, Electrical Engineering and Mathematics
University of Paderborn, Germany
E-mail: belli@adt.upb.de
[2]Siemens Corporation, Corporate Technology, USA
E-mail: christof.budnik@siemens.com
[3]andagon GmbH, Cologne, Germany
E-mail: a.hollmann@andagon.com
[4]Department of Computer Engineering,
Izmir Institute of Technology, Turkey
E-mail: tugkantuglular@iyte.edu.tr
[5]Department of Computer Science, University of Texas at Dallas, USA
E-mail: ewong@utdallas.edu

Abstract—This paper rigorously introduces the concept of model-based mutation testing (MBMT) and positions it in the landscape of mutation testing. Two elementary mutation operators, *insertion* and *omission*, are exemplary applied to a hierarchy of graph-based models of increasing expressive power including directed graphs, event sequence graphs, finite-state machines and statecharts. Test cases generated based on the mutated models (*mutants*) are used to determine not only whether each mutant can be killed but also whether there are any faults in the corresponding system under consideration (SUC) developed based on the original model. Novelties of our approach are: (1) evaluation of the fault detection capability (in terms of revealing faults in the SUC) of test sets generated based on the mutated models, and (2) superseding of the great variety of existing mutation operators by iterations and combinations of the two proposed elementary operators. Three case studies were conducted on industrial and commercial real-life systems to demonstrate the feasibility of using the proposed MBMT approach in detecting faults in SUC, and to analyze its characteristic features. Our experimental data suggests that test sets generated based on the mutated models created by *insertion* operators are more effective in revealing faults in SUC than those generated by *omission* operators. Worth noting is that test sets following the MBMT approach were able to detect faults in the systems that were tested by manufacturers and independent testing organizations before they were released.

Index Terms—mutation testing, model-based testing, model-based mutation testing, mutation operator, insertion operator, omission operator, fault detection capability

1 Introduction

Testing is one of the most commonly used techniques for assuring quality in the software industry. It entails the execution of the software[1] in its real environment, under real-life conditions. The initial step of software development is usually the requirements elicitation, and its outcome is the specification of the system's behavior. It is important to generate test cases and define appropriate testing processes at this early stage, long before the implementation begins, in compliance with the user's expectancy of how the system should behave. An advantage of this is to avoid faults being detected at late stages, when they could have been detected much earlier and fixed less expensively. There are techniques to visualize and represent the relevant features of the system under consideration (SUC) in its environmental context, leading to a *model* of the SUC. These models are increasingly being used as a basis (in the context of model-based testing (MBT)) to generate test cases which satisfy certain criteria, for example, coverage of every node in the model. These tests are then used to validate the SUC. The assumption is that the model is correct, and if the behavior of SUC is consistent with that of the model during conformance testing against test cases generated above, then the SUC must also be correct.

A broad variety of formal or informal models exist for modeling software as recommended in standards such as UML [1] or TTCN-3 [2]. These models describe the SUC at different levels of granularity and preciseness. Graph-based models

*Authors are listed in the alphabetical order of their last names.
[1]The terms "program," "software," "application," and "implementation" are used interchangeably, whenever appropriate.

consist of nodes and arcs. The semantic meaning associated with these nodes and arcs determines the level of precision of the SUC description. The issue of which model to use is not determined solely by identifying which is most suitable for modeling the SUC, but also by taking into account other factors (especially when considering the proposed MBMT approach in Section 3) such as how easily the model can be mutated with respect to certain mutation operators and how test cases can be generated for the model. In this paper, we choose the following formal graph-based models in order of increasing expressive power: (i) Directed graphs (DG) include no semantics; that is, nodes and arcs have no interpretation. The paper uses them to introduce notions, especially mutation operators, syntactically. (ii) Event sequence graphs (ESG) interpret nodes as events and arcs as sequences of events ([3]). (iii) Finite-state machines (FSM) interpret nodes as states, whereas arcs labeled by events symbolising state transitions. (iv) Statecharts (SC) consider concurrency and hierarchy aspects [4,1].

A fundamental problem of testing stems from the huge variety of possible software faults to be considered. For this purpose, *mutation testing* techniques generate faulty versions of a software system (namely, *mutants*) by introducing simple but representative faults using mutation operators (which are also known as mutant operators in the literature). Different strategies have been proposed for mutation testing. Some are applied to source code of the SUC while others are applied to models derived from specifications. Mutants generated can be used at the unit testing [5,6], as well as at the integration testing level [7] (further details on related work are given in Section 6). A common problem of these studies is that they use a large number of mutation operators to generate mutants because they *empirically* determine the mutation operators based on selected fault models. This makes it arguable as to the exact number of operators that should be developed for a given fault model. A different approach is used in this paper by introducing two *elementary* mutation operators: *insertion* and *omission*. Starting with a DG as the underlying model, the elementary operators are applied to other models, such as ESG, FSM, and SC in the same manner. This is also a major difference between this paper and other studies, such as [7,8,9,10,11,12,13,14,15,16]. Many mutation operators known from literature can be reduced to these two elementary operators and their combinations and iterations. For example, the "sdl" operator in the Mothra tool set [17] deletes a statement that is equivalent to an "omission" operator, and the "svr" operator does a scalar variable replacement that is an "omission" followed by an "insertion." In fact, mutants generated by the mutation operators in the aforementioned literature could be viewed as special cases of the fault model developed in this paper. However, mutants generated by the two elementary operators (with multiple iterations) and their combinations presented in this paper might not be generated by the operators introduced by other authors. Refer to Section 4.4 for more details.

The primary objective of this paper is to propose a precise *model-based mutation testing* (MBMT) approach. We view the SUC as a black box and assume that the source code is not available. Hence, it cannot be mutated. Instead, a model (say $\mathcal{M}$) is available and will be mutated. We further assume that the conformance between $\mathcal{M}$ and the corresponding SUC has also been validated by a test set $\mathcal{T}$ created by applying an appropriate test generation algorithm (say Φ) to $\mathcal{M}$. The proposed elementary mutation operators, *insertion* and *omission*, generate a set of mutated models (mutants) when applied to $\mathcal{M}$. The same test generation algorithm Φ which has been applied to $\mathcal{M}$ will be applied to these mutants to generate a test set for each mutant. While using test cases from these sets to determine whether a mutant can be killed, the same test cases can help us detect faults in SUC which have not been revealed during the conformance testing described above using $\mathcal{T}$. In other words, we attempt to assess the fault detection capability of $\mathcal{T}$. This is definitely a different view than the ones found in the literature, e.g.,[18]. Note also in contrast to the code-based mutation testing, mutants in MBMT are classified into more categories as described in other approaches to MBMT. The later sections, especially Section 3, will precisely explain this novel interpretation of MBMT.

Three case studies on industrial and commercial applications covering interactive, reactive, and proactive systems were conducted to demonstrate the feasibility of using the proposed MBMT approach and to discuss its characteristic features. The results of our experiments suggest that tests generated by the proposed approach can be effective in detecting faults in the corresponding SUC. For example, in the third case study (the control terminal of a marginal strip mower), the test cases generated by the approach described here could detect faults in the released version that were not found by the previous (and independent) testers, such as a technical control board in Germany.

The remainder of the paper is organized as follows. Section 2 summarizes MBT and reviews the models and test generation processes selected for our study. Section 3 discusses MBMT and explains the difference between this new approach and the code-based mutation testing. Section 4 defines the elementary mutation operators and compares them with other mutation operators reported in the literature. Section 5 presents case studies including results, analysis and discussion. Section 6 gives related work. Our conclusion and future work appear in Section 7.

2 Modeling and Model-Based Testing

In modeling and model-based testing (MBT), features of an SUC are described by a model $\mathcal{M}$ that is constructed in accordance with the specification (note that specification-based testing does not necessarily require a model). As mentioned in the introduction, it is assumed that $\mathcal{M}$ is correct, and its correctness has been assured, for example, by using model checking techniques [19] with respect to some properties derived from the requirements. A *test generation algorithm* Φ using a test selection criterion [20], for example, covering every node if $\mathcal{M}$ is graph-based, is applied to $\mathcal{M}$ such that $\Phi(\mathcal{M})$ generates a *test set* $\mathcal{T} = \{t_1, t_2, ..., t_n\}$ with each t_i (i=1, ..., n) being a *test case* as an ordered pair t_i = (*input to the SUC, expected output from the SUC*) $\in \mathcal{T}$. If no failures are observed when SUC is executed against $\mathcal{T}$, we conclude that SUC is consistent with $\mathcal{M}$ (with respect to $\mathcal{T}$). Since we assume $\mathcal{M}$ is correct, we may consequently assume SUC is correct. However, this assumption may not extend to test cases that are not in $\mathcal{T}$. Hence, even if the execution of $\mathcal{T}$ does not reveal any faults in SUC, it does not imply SUC is correct. This leads to an important question: how good is $\mathcal{T}$ in terms of detecting faults? To answer this question, we need to have a way to assess the fault detection capability of $\mathcal{T}$. To exemplify our approach, a summary of our models is presented below.

Definition 1. *A directed graph DiGraph* = (N, A) is an ordered pair where N is a finite set of nodes and $A \subseteq N \times N$ is a set of arcs.

In Definition 1, the nodes and arcs do not have any semantics.

2.1 Event Sequence Graphs

For exemplifying modeling, model-based testing, and model mutation, we interpret the nodes of a DG as events and arcs as sequences of events leading to *event sequence graphs* (*ESG*) [3]. Generally, an *event* represents an externally observable phenomenon, such as an environmental or a user stimulus, or a system response punctuating different stages of the system activity of an SUC, which is assumed to be deterministic. ESG are commonly used for analysis and validation of user interface requirements and are similar to the concept of event flow graphs [21]. We choose ESG because it intensively uses formal graph-theoretical notions and algorithms [22].

Definition 2. An *event sequence graph ESG* = (DG, Ξ, Γ) is a directed graph, where

- DG = (E, A) is a directed graph as defined in Definition 1 with E being a finite set of *events* and $A \subseteq E \times E$ being a set of *arcs*
- Ξ (*entry* events) $\subseteq E$ and Γ (*exit* events) $\subseteq E$ are sets of *distinguished* events with $|\Xi| \geq 1$ and $|\Gamma| \geq 1$

The syntax of a *valid* ESG requires that every event has to be reached by an entry and that from every event an exit has to be reached; otherwise the ESG is *invalid*. To identify the entry and exit events of an ESG graphically, every $\xi \in \Xi$ is preceded by a pseudo-event "[" $\notin E$ and every $\gamma \in \Gamma$ is followed by a pseudo-event "]" $\notin E$. Pseudo-event represented by node "[" $\notin E$ and pseudo-event represented by node "]" $\notin E$. Pseudo-events are annotations and do not belong to DG. The semantics of the arcs of an ESG are as follows. For two events, ψ and $\psi' \in E$, ψ' must be enabled after the execution of ψ if and only if $(\psi, \psi') \in A$. It is assumed that a DG has to be strongly connected. Even if at most one arc in A is missing to have the graph strongly connected, the graph is still valid. An example of an ESG is given in Fig. 1 that models a simple CD player.

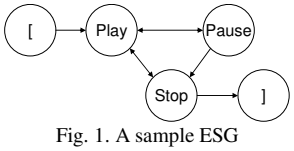

Fig. 1. A sample ESG

In Fig. 1, the DG = (E, A), where E = {Play, Pause, Stop} and A = {<Play,Pause>, <Play,Stop>, <Pause,Play>, <Pause,Stop>, <Stop,Play>}. DG of the sample ESG given in Fig. 1 is strongly connected.

Definition 3. Let E and A be the finite set of events and arcs of an ESG. Any sequence of events (e_1, ..., e_k) with $e_i \in E$ for i=1, ..., k is an *event sequence* (ES) of length k, if $(e_i, e_{i+1}) \in A$ for i=1, ..., k–1.

Definition 4. An ES = (e_1, ..., e_k) is a *complete event sequence* (*CES*) if $e_1 \in \Xi$ and $e_k \in \Gamma$.

Each CES represents a *walk* from the entry of an ESG to its exit, realized by a chain of user inputs and system responses. A test case is an ordered pair of an input and an expected output of the SUC. A test set can contain any number of test cases. A CES of an ESG can be used as a test case for the corresponding SUC. The test cases are thereby formed by the events, particularly by the user activities (inputs) and the expected system responses (outputs). A test execution with respect to a CES *fails* if and only if the CES cannot reach the corresponding exit event due to a failure, such as a system crash or if it reaches an exit event but does not produce the expected outputs.

Fig. 2 presents a coverage-based test generation process to produce a set of CES covering all event sequences of a required length in a given ESG. Let the *length* of each test case (each CES) be the number of events it has. The total length of a test set is the sum of the lengths of all its test cases. To minimize this length, solutions of the *Chinese Postman Problem* can be used; that is, finding the shortest path or circuit in a graph by visiting each arc. Polynomial algorithms supporting this test generation process have been published before, e.g., in our previous work [22]. The reason for such minimization is to reduce the cost of test execution.

Input: An ESG and a number $n \in \mathbb{N}$ **Output**: A test set $\mathcal{T} = \{t_1, t_2, ..., t_m\}$ to cover all event sequences of length n. Each test case in $\mathcal{T}$ is a CES.
Generate a set of CES to cover all event sequences of length n in the ESG such that $\sum_{i=1}^{m} length(t_i)$ is minimal

Fig. 2. An ESG-based test generation process (details of the algorithms can be found in our previous work [22])

Referring to the ESG in Fig. 1, we have event sequence sets of different lengths. For example:

ES_2 = {event sequences of length 2}
= {(Play, Pause), (Pause, Play), (Play, Stop), (Stop, Play), (Pause, Stop)}.

To cover all the sequences in ES_2, a test set

$\mathcal{T}$ = {(Play, Pause, Play, Pause, Stop, Play, Stop)}

with one CES is generated using the process in Fig. 2. $\mathcal{T}$ has a total length of 7, which is the minimum among all the test sets that can cover the five event sequences in ES_2. Similarly, we have

ES_3 = {event sequences of length 3}
= {(Play, Stop, Play), (Play, Pause, Play), (Play, Pause, Stop), (Pause, Play, Stop), (Pause, Play, Pause), (Pause, Stop, Play), (Stop, Play, Stop), (Stop, Play, Pause)}.

To cover all the sequences in ES_3, another test set $\mathcal{T}$' is generated with two CES as follows

$\mathcal{T}$' = {(Play, Pause, Play, Pause, Stop, Play, Pause, Stop), (Play, Stop, Play, Stop, Play, Pause, Play, Stop)}

$\mathcal{T}$' has a total length of 16 (=8+8), which is the minimum among all the test sets that can cover all the event sequences in ES_3.

2.2 Finite-State Machines and Statecharts

To model more complex systems using states, hierarchy, concurrency, etc., we exemplary[2] choose FSM and statecharts. Based on the notions of DG, statecharts [4] are defined as follows. Note that the following definition of statecharts also includes the definition of FSM.

Definition 5. A *basic statechart* is given by $SC = (FSM, H, g)$, where

- $FSM = (DG, E, f, \Xi, \Gamma)$ is a finite-state machine
  - $DG = (S, TR)$ as defined in Definition 1 with S being a set of states and $TR \subseteq S \times S$ being a set of transitions
  - E is a finite set of events
  - f: $TR \rightarrow E$ is a function, mapping an event to each transition
  - $\Xi \subseteq S$ and $\Gamma \subseteq S$ are finite sets of initial and final states
- $H \subseteq S \times S$ is a hierarchy relation
- g: $S \rightarrow \{S_{simple}, S_{and}, S_{xor}\}$ is a total labeling function

[2] Other models (such as formal grammars, regular expressions, and various UML diagrams) can also be used.

An FSM or SC can be viewed as a DG with semantically distinguished states (nodes) and state transitions (arcs). Nodes represent states. Arcs, labeled by events, are transitions between states. In our study, only deterministic FSM and SC are considered. We also require that each state can be reached from the initial state and a final state is reachable from each state. The set H is used to represent the hierarchy relation between states in statecharts. Each state $s \in S$ is identified as a simple state (S_{simple}) or a composite state, and the latter is further classified as an AND-state (S_{and}) or an XOR-state (S_{xor}). The sub-states of an AND-state are called *regions*. It means that the statechart resides concurrently in each region of the AND-state.

Definition 6. A sequence of transitions (tr_1, ..., tr_k) with $tr_i \in TR$ for i=1, ..., k (where TR is defined in Definition 5) is a *transition sequence* (TS) of length k, if (tr_i=(s_α, s_β), tr_{i+1}=(s_γ, s_δ)) denotes a pair of transitions for i=1, ..., k–1 with $s_\beta = s_\gamma$.

Definition 7. A TS= (tr_1, ..., tr_k) is a *complete transition sequence* (CTS) if it starts at the initial and ends at a final state.

Fig. 3 gives a coverage-based test generation process for FSM and SC to generate a set of CTS covering all transition sequences of a required length. Similar to the process for ESG (see Fig. 2), each CTS can be used as a test case and the total length of all these CTS is minimized to reduce the cost of test execution. Detailed algorithms can be found in [4]. Note that each CTS = (tr_1, ..., tr_k) is mapped to a corresponding CES = (e_1, ..., e_k) where e_i=f(tr_i) for i =1, ..., k.

Input: An FSM or SC and a number $n \in \mathbb{N}$ **Output**: A test set $\mathcal{T}$= {t_1, ..., t_m} to cover all transition sequences of length n. Each test case in $\mathcal{T}$ is a CTS.
Generate a set of CTS to cover all transition sequences of length n in the FSM or SC such that $\sum_{i=1}^{m} length(t_i)$ is minimal. Map each CTS to the corresponding CES

Fig. 3. An FSM and SC-based test generation process (for more details see [4])

3 Model-Based Mutation Testing (MBMT)

This section discusses the approach for conducting MBMT, followed by a discussion of how to classify (Section 3.1), generate, and execute (Section 3.2) mutants (namely, mutated models).

Mutation testing, as introduced in the late seventies of the last century, is a fault injection-based white-box testing technique [23,24,25] (see Fig. 4). It relies on two hypotheses: the *competent programmer hypothesis* [26,27] and the *coupling effect* [28]. The competent programmer hypothesis suggests that experienced programmers tend to write programs that are "close" to being correct. That is, although a program written by a competent programmer may be incorrect, it will differ from the correct version by only relatively simple faults in terms of their syntax and semantics. The task of testing is therefore reduced to validating a program that is probably not correct but is close to the correct one. The coupling effect assumes that a test set that detects all simple faults in a program is also likely to detect more complex faults [28]. Mutants generated by introducing exactly one change into a program are known as *first-order* mutants [28]. *Second-order* mutants are generated by making two simple changes. Mutants with more than one change are also known as *higher-order* mutants. According to the coupling effect, higher-order mutants are likely to be detected by test cases that detect first-order mutants. In general, therefore, only first-order mutants are generated and used in mutation testing.

For explanatory purposes, assume that a program $\mathcal{P}$ and a test set $\mathcal{T}$ are given. A set of mutation operators $\mathcal{X}$ is then applied to $\mathcal{P}$ to generate mutants by introducing one or more syntactical changes into $\mathcal{P}$. Based on the coupling effect, the focus is on the first-order mutants. A mutant is *killed* ("distinguished") by a test case t in $\mathcal{T}$ if the observed behavior of $\mathcal{P}$ differs from that of the mutant when executed against t. A mutant is *equivalent* to $\mathcal{P}$ if it always behaves the same as $\mathcal{P}$ for every case (i.e., every possible input). The test set $\mathcal{T}$ is said to be *mutation adequate* [20] with respect to $\mathcal{P}$ and $\mathcal{X}$ if every non-equivalent mutant of $\mathcal{P}$ generated by applying the mutation operators in $\mathcal{X}$ can be killed by at least one test case in $\mathcal{T}$. Mutation testing has also been extended to validate a specification $\mathcal{S}$. To do this, a set of mutation operators is applied to $\mathcal{S}$ to generate mutated specifications ($\mathcal{S}^*$). The adequacy of $\mathcal{T}$ is measured by how many non-equivalent $\mathcal{S}^*$ can be killed by test cases in $\mathcal{T}$. Fig. 4 depicts the above discussion. Such mutation testing (hereafter referred to as "code-based mutation testing") is not always feasible because, for example, the program source may not be available.

The MBMT proposed in this paper assumes that (i) the conformance of $\mathcal{M}$ and the corresponding SUC has already been validated a priori by model-based testing (MBT) using a test set $\mathcal{T}$, and (ii) there may be still latent faults in SUC that were not revealed by $\mathcal{T}$. Referring to Fig. 5, the solid lines are for the MBT and the dashed lines for the follow-on MBMT. Let the *set of mutation operators* be defined as $\mathcal{X}$ = {χ_1, ..., χ_i, ..., χ_k}. Applying an operator χ_i to $\mathcal{M}$ (denoted by $\chi_i(\mathcal{M})$) gener-

ates a *set of mutated models* $\mathcal{M}^*_i = \{\mathcal{M}^*_{i1}, \mathcal{M}^*_{i2}, ..., \mathcal{M}^*_{ij}, ..., \mathcal{M}^*_{ip}\}$, where $p \geq 1$. These mutants are called *model mutants* of $\mathcal{M}$. For the rest of the paper, they are simply referred to as *mutants* when there is no ambiguity. From each $\mathcal{M}^*_{ij}$ (the j^{th} mutant generated by the i^{th} operator), a test set $\mathcal{T}^*_{ij}$ is generated based on a test generation algorithm Φ, that is, $\Phi(\mathcal{M}^*_{ij}) = \mathcal{T}^*_{ij} = \{t^*_{ij_1}, t^*_{ij_2}, ..., t^*_{ij_n}\}$. For explanatory purposes, we use $\mathcal{M}^*$ and $\mathcal{T}^*$ as a generic representation for a mutant generated from $\mathcal{M}$ and a test set with respect to $\mathcal{T}^*_{ij}$, respectively. The objective of MBMT is to use all the $\mathcal{T}^*$ to help us detect possible additional bugs in SUC which cannot be revealed previously by $\mathcal{T}$ (generated by applying the same test generation algorithm Φ to the model $\mathcal{M}$). Note that we assume the competent programmer hypothesis and the coupling effect also apply to MBMT. This is recognized as a potential threat to the validity of our approach (see Section 5.44).

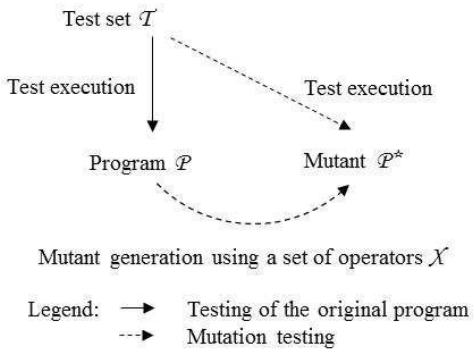

Fig. 4. Code-based mutation testing

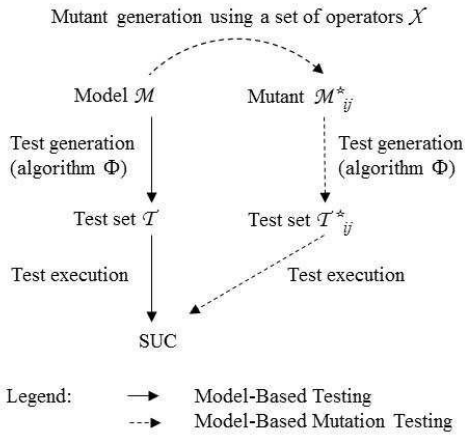

Fig. 5. Model-Based Mutation Testing

3.1 Mutant Classification for MBMT

The following explains how to determine whether $\mathcal{M}^*$ is a valid or an invalid mutant of $\mathcal{M}$, and whether $\mathcal{M}^*$ has been *killed* or is still *live* with respect to test cases in $\mathcal{T}^*$, or whether or not $\mathcal{M}^*$ is equivalent to $\mathcal{M}$.

Definition 8. *valid* $(\mathcal{M}^*) := \{\mathcal{M}^* \mid \mathcal{M}^*$ satisfies the syntactic and semantic requirements imposed by the model type of $\mathcal{M}\}$

For example, if $\mathcal{M}$ is an FSM, then $\mathcal{M}^*$ (a mutated model) must also be a valid FSM. A syntactical requirement for an FSM is that all transitions must be associated with states. Semantics might require that an FSM must be deterministic or that all states are reachable from an initial state.

Definition 9. *killed* $(\mathcal{M}^*, \text{SUC}, \Phi) := valid\ (\mathcal{M}^*) \wedge \exists\ t^* \in \Phi(\mathcal{M}^*), Output(\text{SUC}, t^*) \neq ExpectedOutput\ (t^*)$

If there is at least one test case t^* in $\mathcal{T}^*$ such that the outputs of $\mathcal{M}^*$ and SUC are different (which is also the output of $\mathcal{M}$ as the conformance between SUC and $\mathcal{M}$ has been validated already by $\mathcal{T} = \Phi(\mathcal{M})$ as shown in Fig. 5), then $\mathcal{M}^*$ is killed by t^*. Note that *ExpectedOutput* (t^*) is the output of $\mathcal{M}^*$ on t^*.

Definition 10. *equivalent* $(\mathcal{M}, \mathcal{M}^*) := \forall\ t$ in the input domain of $\mathcal{M}$, $Output(\mathcal{M}, t) = Output(\mathcal{M}^*, t)$

A mutant $\mathcal{M}^*$ is *equivalent* to $\mathcal{M}$ if and only if on every element of their input domain $\mathcal{M}^*$ and $\mathcal{M}$ produce the same outputs.

Definition 11. *live* $(\mathcal{M}^*, \text{SUC}, \Phi) := valid\ (\mathcal{M}^*) \wedge \forall\ t^* \in \Phi(\mathcal{M}^*), Output\ (\text{SUC}, t^*) = ExpectedOutput\ (t^*)$

A mutant $\mathcal{M}^*$ is *live* with respect to $\mathcal{T}^* = \Phi(\mathcal{M}^*)$ if and only if no test case t^* in $\mathcal{T}^*$ can kill $\mathcal{M}^*$.

With respect to the code-based mutation testing in Fig. 4, a non-equivalent mutant is either *killed* (i.e., the injected fault in $\mathcal{P}^*$ is revealed) or *live* (i.e., the fault in $\mathcal{P}^*$ cannot be detected) with respect to a given test set $\mathcal{T}$. However, the classification of mutants for MBMT is much more complicated. For example, a test case t^* killing a mutant $\mathcal{M}^*$ does not necessarily imply that t^* detects the fault injected into $\mathcal{M}^*$. Based on Definition 9, $\mathcal{M}^*$ is killed because *Output*(SUC, t^*) differs from *Ex-*

pectedOutput (t^*). However, such difference may be due to a fault in SUC which was not detected previously when we validated the conformance between $\mathcal{M}$ and SUC using the test set $\mathcal{T}$=$\Phi(\mathcal{M})$. Under this condition, t^* helps us find a fault in SUC which has not been detected by $\mathcal{T}$ instead of revealing the injected fault in $\mathcal{M}^*$. Fig. 6 gives the complete classification.

A mutant $\mathcal{M}^*$ can be *live* for one of the following reasons. First, it is possible that $\mathcal{M}^*$ is equivalent to $\mathcal{M}$ (see Box (a) in Fig. 6). Second (see Box (b)), $\mathcal{T}^*$ is unable to detect the previously injected fault in $\mathcal{M}^*$. There are two possible reasons. Test cases in $\mathcal{T}^*$ do not execute the mutated part of $\mathcal{M}^*$ or the expected output of $\mathcal{T}^*$ is unable to reveal the difference between $\mathcal{M}$ and $\mathcal{M}^*$. Third (see Box (c)), it is possible that SUC behaves the same as $\mathcal{M}^*$ (which is not equivalent to $\mathcal{M}$) with respect to test cases in $\mathcal{T}^*$. This indicates that the implementation of SUC deviates from model $\mathcal{M}$ even though a conformance is established between SUC and $\mathcal{M}$ with respect to $\mathcal{T}$=$\Phi(\mathcal{M})$. This also implies that SUC contains "code" which introduces some "unexpected" behavior – different from that specified by $\mathcal{M}$. This helps testers detect additional faults in SUC that have not been revealed by $\mathcal{T}$=$\Phi(\mathcal{M})$. Note that for the code-based mutation testing, we only have the first two cases (namely, Boxes (a) and (b)), but not the third case (Box (c)).

When a mutant $\mathcal{M}^*$ is *killed* by $\mathcal{T}^*$, then SUC behaves differently from $\mathcal{M}^*$ on at least one test case $t^* \in \mathcal{T}^*$ (see Definition 9). There are three possibilities. First (see Box (d)), the injected fault in $\mathcal{M}^*$ is not in SUC. When SUC is executed against t^*, it does not show certain faulty behavior of $\mathcal{M}^*$. Second (see Box (e)), SUC has some faults that are not in $\mathcal{M}^*$ nor in $\mathcal{M}$. This is possible because the conformance between SUC and $\mathcal{M}$ is only validated by $\mathcal{T}$=$\Phi(\mathcal{M})$ rather than every possible input. As a result, faults in SUC may cause its execution against a test case t^* to produce an output differing from the expected output of $\mathcal{M}^*$, which does not contain the same fault. Third (see Box (f)), the injected fault in $\mathcal{M}^*$ is also in SUC, but not in $\mathcal{M}$. This is possible for the same reason described above. However, although the fault is the same in SUC and $\mathcal{M}^*$, it will manifest in different ways, which makes the output of SUC differ from the expected output of $\mathcal{M}^*$ for at least one test case $t^* \in \mathcal{T}^* = \Phi(\mathcal{M}^*)$. Note that for the code-based mutation testing, we only have the first case (namely, Box (d)), but not the last two case (Boxes (e) and (f)).

Interpretation of whether a live mutant found a bug or not is performed at line 9 of Algorithm 1, where Output(SUC, t^*) is checked for being equal to ExpectedOutput(t^*). If that check is TRUE for all $t^* \in \mathcal{T}^*_{ij}$, then $\mathcal{M}^*_{ij}$ is a live mutant. Then the classification tree in Fig. 6 is used. $\mathcal{M}^*_{ij}$ is checked for equivalence to $\mathcal{M}$.

1. If they are equivalent, then (since we assumed that SUC behaves the same as $\mathcal{M}$) we conclude that SUC behaves the same as $\mathcal{M}^*_{ij}$. That means test set $\mathcal{T}^*_{ij}$ did not find a new fault in SUC.
2. If they are not equivalent, then there are two possibilities:
   i. SUC behaves the same as $\mathcal{M}^*_{ij}$, which is (c) leaf of classification tree in Fig.6.
   ii. $\mathcal{T}^*$ is unable to detect the previously injected fault in $\mathcal{M}^*$. There are two possible reasons. Test cases in $\mathcal{T}^*$ do not execute the mutated part of $\mathcal{M}^*$ or the expected output of $\mathcal{T}^*$ is unable to reveal the difference between $\mathcal{M}$ and $\mathcal{M}^*$, which is (b) leaf of classification tree in Fig. 6.

To determine which of the above two possibilities, i.e. (2.i) and (2.ii), occurred, $\mathcal{M}$ is tested with test set $\mathcal{T}^*_{ij}$. If Output($\mathcal{M}$, t^*) being equal to ExpectedOutput(t^*) is TRUE for all $t^* \in \mathcal{T}^*_{ij}$, then (since we assumed that SUC behaves the same as $\mathcal{M}$) SUC behaves the same as $\mathcal{M}^*_{ij}$. If Output($\mathcal{M}$, t^*) being equal to ExpectedOutput(t^*) is not TRUE for all $t^* \in \mathcal{T}^*_{ij}$, then SUC behaves different than $\mathcal{M}^*_{ij}$.

In summary, Boxes (a), (b), and (d) are the code-based interpretation of *killed* and *live* mutants with respect to $\mathcal{M}^*$ and $\mathcal{T}^*$. On the other hand, Boxes (c), (e), and (f) in Fig. 6 represent cases where faults in SUC can be detected by MBMT. Stated differently, if there are any mutants in these categories, then SUC has some faults that are not in $\mathcal{M}$ and cannot be detected by the test set T generated by applying a test generation algorithm Φ to the model $\mathcal{M}$. This also implies that if $|LM_{ne1}| + |KM_{na1}| + |KM_{na2}| \neq 0$, then $\mathcal{T}$ should be improved as it fails to detect some faults in SUC. Also, if there are any mutants in Box (b) (i.e., LM_{ne2} is not empty), then some non-equivalent mutants are still *live*, but may be *killed* if $\mathcal{T}^* = \Phi(\mathcal{M}^*)$ is improved with additional test cases. Each of these cases (either $\mathcal{T}$ or $\mathcal{T}^*$ should be improved) suggests that the corresponding test generation algorithm Φ should be improved. Based on the above discussion, we define the following scores.

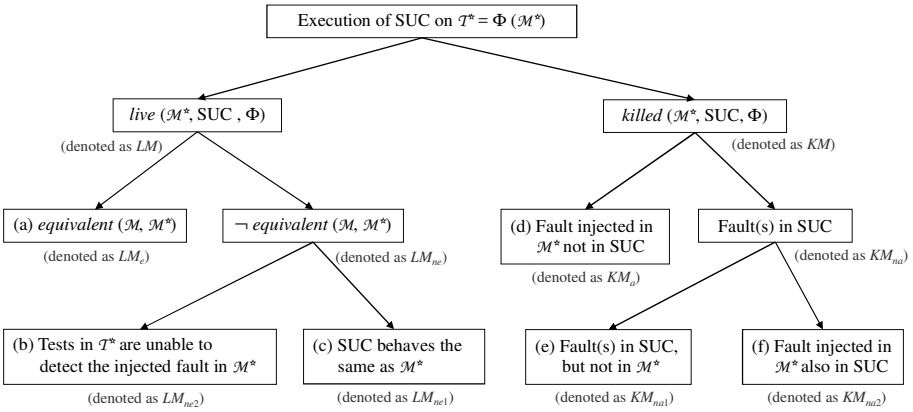

Fig. 6. Mutant classification (arcs represent "or")

Definition 12. Given a model $\mathcal{M}$, the corresponding SUC, a test generation algorithm Φ, and a set of mutation operators $\mathcal{X}$, the *mutation fault detection score* $MFDS(\mathcal{M}, \mathrm{SUC}, \Phi, \mathcal{X}) := \frac{|LM_{ne1}| + |KM_{na1}| + |KM_{na2}|}{|\mathcal{M}^*| - |LM_e|} = \frac{|LM_{ne1}| + |KM_{na}|}{|\mathcal{M}^*| - |LM_e|}$, where $|\mathcal{M}^*|$ is the number of all the mutants generated.

MFDS gives the ratio of the number of mutants identified as *killed* or *live* that help detect faults in SUC (the numerator) to the total number of non-equivalent mutants (the denominator). Refer to Section 5.2 for more explanation.

Definition 13. Given a model $\mathcal{M}$, the corresponding SUC, a test generation algorithm Φ, and a set of mutation operators $\mathcal{X}$, the *modified mutation score* $MMS(\mathcal{M}, \mathrm{SUC}, \Phi, \mathcal{X}) := \frac{|KM_a|}{|\mathcal{M}^*| - |LM_e|}$, where $|\mathcal{M}^*|$ is the number of all the mutants generated.

MMS, similar to the code-based mutation score, gives the ratio of the mutants that are killed (but do not help reveal any faults in SUC) to the total number of non-equivalent mutants. Assuming there is at least one non-equivalent mutant, *MMS* has a value of 1 if $|LM_{ne1}| + |KM_{na1}| + |KM_{na2}| = 0$ (i.e., MBMT does not help the detection of any additional faults in SUC which have not been revealed by test cases in $\mathcal{T} = \Phi(\mathcal{M})$) and $|LM_{ne2}| = 0$ (i.e., no mutants are live where the reason of liveness is because $\mathcal{T}^*$ cannot detect the injected faults).

3.2 Mutant Generation and Execution for MBMT

In the following, a model-independent algorithm (Algorithm 1) for mutant generation, execution of SUC, and classification of mutants into *killed* or *live* is presented. The input consists of a model $\mathcal{M}$ that describes the underlying SUC, a set of mutation operators $\mathcal{X}$, and a test generation algorithm Φ. The output is a list of mutants generated by the mutation operators in $\mathcal{X}$ that are *killed* by test cases generated by $\Phi(\mathcal{M}^*)$ and a list of mutants that are still *live*. Each mutation operator $\chi_i \in \mathcal{X}$ is applied to $\mathcal{M}$ to generate a set of mutants denoted as $\mathcal{M}^*_i$. For each valid mutant $\mathcal{M}^*_{ij} \in \mathcal{M}^*_i$, SUC is executed against test cases (one by one) in the corresponding $\mathcal{T}^*_{ij}$ generated by applying the algorithm Φ to $\mathcal{M}^*_{ij}$. The mutant $\mathcal{M}^*_{ij}$ is marked as *killed* if the expected output of a test case t^* differs from that of SUC. Note that in future work we will examine the conditions under which the execution of remaining $t^* \in \mathcal{T}^*_{ij}$ is necessary. If no test case in $\mathcal{T}^*_{ij}$ can kill $\mathcal{M}^*_{ij}$, the mutant is marked as *live*. Details of $\Phi(\mathcal{M})$ for event and state-based models such as ESG (event sequence graph), FSM (finite-state machine) and SC (statecharts) have been presented in Section 2.

4 Elementary Mutation Operators

The proposed MBMT approach is exemplified using the graph-based models introduced in Section 3. We explain in detail the two proposed elementary mutation operators (*insertion* and *omission*) and their applications.

Algorithm 1. Mutant generation, execution and classification

Input: Model $\mathcal{M}$ that describes SUC
Set of mutation operators $\mathcal{X} = \{\chi_1, ..., \chi_k\}$
Test generation algorithm Φ
Output: A list of killed and live mutants

```
1   foreach χ_i ∈ X
2   {  M*_i = χ_i(M)                          // apply the i^th mutation operator χ_i to M
3      foreach M*_ij ∈ M*_i
4      {   if valid(M*_ij) then                // check the validity of each mutant
5          {  T*_ij := Φ (M*_ij)               // test case generation
6             killed := false
7             foreach t* ∈ T*_ij
8             {  Execute SUC against t*
9                if Output(SUC, t*) ≠ ExpectedOutput(t*) then
10               {  Add M*_ij to the list of killed mutants
11                  killed := true
12                  continue                   // as soon as a test case t* kills the mutant M*_ij ,
13                                             // the remaining test cases in T*_ij are executed.
14               }
15            }
16            if killed = false then
17               Add M*_ij to the list of live mutants
18         }
19      }
20  }
```

4.1 Syntactic Definition based on Directed Graphs

In the following, the two proposed elementary mutation operators are syntactically introduced based on directed graphs; their basic features are also discussed. Two types of mutants (*node* and *arc* mutants) can be generated for each *DG*. For each type, two elementary mutation operators are defined, namely *insertion* and *omission*. To generate a first-order mutant, a mutation operator is applied exactly once to a DG. Note that even in this initial step, the mutated $DG^* = (N^*, A^*)$ may become invalid (e.g., unconnected). Only in the case that an operator would always generate invalid mutants, a minimum set of additional operations are also applied to make the mutants valid. Higher-order mutants can be generated by applying the elementary operators or their combinations, multiple times.

Definition 14. The *node insertion* operator $nI(DG, \alpha) \rightarrow DG^* (N \cup\{\alpha\}, A \cup \{(\beta, \alpha) \cup (\alpha, \gamma)\})$, where α, β and γ are three nodes such that $\alpha \notin N$, β and $\gamma \in N$.

The *node insertion* operator (*nI*-operator) inserts a node $\alpha \notin N$ into a given DG. To keep the resulting graph strongly connected, it also inserts an incoming arc to α, say (β, α), and an outgoing arc from α, say (α, γ), where β and γ are arbitrarily chosen from N such that $\beta \neq \alpha$ and $\alpha \neq \gamma$.

Definition 15. The *node omission* operator $nO(DG, \alpha) \rightarrow DG^* (N \setminus \{\alpha\}, A \setminus \{(\beta, \alpha) \mid (\beta, \alpha) \in A, \alpha \text{ and } \beta \in N\} \setminus \{(\alpha, \gamma) \mid (\alpha, \gamma) \in A, \alpha \text{ and } \gamma \in N\})$.[3]

The *node omission* operator (*nO*-operator) removes a node $\alpha \in N$ from a given DG. It also removes the arcs that have α as either the starting or ending node. If the resulting graph violates the validity of DG (e.g., becoming unconnected), it is *discarded*.

Definition 16. The *arc insertion* operator $aI(DG, (\alpha, \beta)) \rightarrow DG^* (N, A \cup \{(\alpha, \beta)\})$, where α and $\beta \in N$ but $(\alpha, \beta) \notin A$.

The *arc insertion* operator (*aI*-operator) inserts an arc $(\alpha, \beta) \notin A$ between nodes α and β in the given DG.

Definition 17. The *arc omission* operator $aO(DG, (\alpha, \beta)) \rightarrow DG^* (N, A \setminus \{(\alpha, \beta)\})$, where $(\alpha, \beta) \in A$.

[3] The notion "\" represents "set subtraction".

The *arc omission* operator (*aO*-operator) removes an existing arc (α, β) from a given DG. If the resulting graph becomes invalid (unconnected), it is discarded.

Definition 18. A *subgraph DG'* of a directed graph *DG* is defined as $DG' \subset DG := ((N' \subset N \wedge A' \subseteq A) \vee (N' \subseteq N \wedge A' \subset A))$. If two graphs DG_1 and DG_2 are not subgraphs of each other, it is written as $DG_1 \rhd\lhd DG_2$.

Using Definition 18 we define increscent, decrescent and cross mutants as follows.

Definition 19.

- DG^* is an *increscent mutant* (*I*-mutant) of *DG* if $DG \subset DG^*$
- DG^* is a *decrescent mutant* (*D*-mutant) of *DG* if $DG^* \subset DG$
- DG^* is a *cross mutant* (*C*-mutant) if $DG \rhd\lhd DG^*$

A given DG is a subgraph of its increscent mutants, and any of its decrescent mutants are subgraphs of the original DG. There is no subgraph relation between a DG and its cross mutants. Using the notions of Definition 19, we have following lemma.

Lemma 1.

- Mutants DG^*, generated by the *nI* and *aI*-operators, are *I*-mutants.
- Mutants DG^*, generated by the *nO* and *aO*-operators, are *D*-mutants.
- *C*-mutants can only be generated by applying an insertion operator (*nI* or *aI*) followed by an omission operator (*nO* or *aO*), or vice versa. Each operator can be applied multiple times, if necessary.

The first two parts of the Lemma are straightforward. The third part says to generate a *C*-mutant of a DG, we not only insert "a node or an arc" into the DG but also delete "a node or an arc different from that being inserted" from the DG. Whether the insertion is before or after the deletion is irrelevant. For example, a *C*-mutant can be generated by first applying the *nI-operator* to insert a node (say α) and the associated arcs, followed by the *nO*-operator to delete a node (different from α) and the associated arcs from the DG. Similarly, a *C*-mutant can also be generated by applying the *aO*-operator to delete an arc (say (α, β)) followed by the *aI*-operator to insert an arc (different from (α, β)).

Example 1. Mutants in Fig. 7, Fig. 8 and Fig. 9 give examples of an *I*-mutant, a *D*-mutant and a *C*-mutant of the ESG in Fig. 1, respectively.

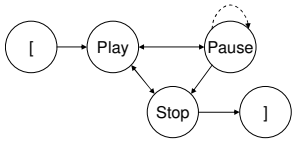

Fig. 7. An *I*-mutant of the ESG given in Fig. 1

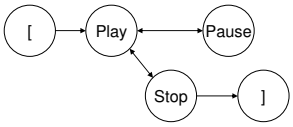

Fig. 8. A *D*-mutant of the ESG given in Fig. 1

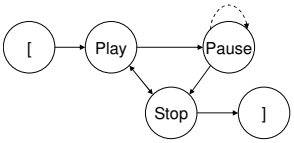

Fig. 9. A *C*-mutant of the ESG given in Fig. 1

A problem with mutant generation is that the application of different mutation operators can lead to the same mutants. When applying the proposed MBMT, test cases are generated with respect to each mutant (see Algorithm 1 in Section 3.2). Hence, multiple occurrences of any duplicate mutants introduce inefficiency and unnecessary waste of testing resources. We now discuss whether there are any duplicates among the first-order mutants generated by applying the *nI*, *aI*, *nO*, or *aO*-operator exactly once to a given DG. Let DG^*_{nI}, DG^*_{aI}, DG^*_{nO}, and DG^*_{aO} be the mutants generated by the *nI*, *aI*, *nO,* and *aO*-operators, respectively. Also, let A^*_{nI}, A^*_{aI}, A^*_{nO} and A^*_{aO} be the resulting set of arcs due to the application of the *nI*, *aI*, *nO*, and *aO*-operators, respectively, and N^*_{nO} and N^*_{aO} be the resulting set of nodes due to the application of the *nO* and *aO*-operators, respectively.

Theorem 1. First-order mutants of a DG generated by the *nI*, *aI*, *nO,* and *aO*-operators are different from each other.

Proof:

- *nI* versus *nO*, *nI* versus *aO*, *aI* versus *nO*, and *aI* versus *aO*: from Lemma 1 it follows that mutants are different.
- *nI* versus *aI*: $DG^*_{nI} \rhd\lhd DG^*_{aI}$ as $(A^*_{nI} \not\subset A^*_{aI}) \wedge (A^*_{aI} \not\subset A^*_{nI})$
  Hence, first-order mutants generated by the *nI*-operator are different from those generated by the *aI*-operator.

- *nO*(*DG*, α) versus *aO*(*DG*, (β, γ)): if (α=β ∨ α=γ) then $DG^*_{nO} \subset DG^*_{aO}$ as $(N^*_{nO} \subset N^*_{aO}) \wedge (A^*_{nO} \subset A^*_{aO})$ else $DG^*_{nO} \rhd\lhd DG^*_{aO}$ as $(A^*_{nO} \not\subset A^*_{aO}) \wedge (A^*_{aO} \not\subset A^*_{nO})$
  Hence, first-order mutants generated by the *nO*-operator are different from those generated by the *aO*-operator ■

The *node corruption* operator (*nC*-operator) first deletes a node from a given DG, then inserts another node which is not currently in the DG. It can be viewed as the application of an *nO*-operator followed by an *nI*-operator. The *arc corruption* operator (*aC*-operator) changes the direction of an existing arc of a given DG. It can be viewed as the application of an *aO*-operator followed by an *aI*-operator. The *nI*, *nO*, *aI*, *aO*, *nC* and *aC*-operators can be applied repeatedly to a DG multiple times. They can also be combined with each other.

4.2 ESG-Based Mutation

If the underlying model in the proposed MBMT approach is an ESG, it is called an ESG-based mutation. Let us assume an ESG correctly specifies the expected behavior of the corresponding SUC. This ESG can then be used to generate mutants $ESG^* = (E^*, A^*, \Xi^*, \Gamma^*)$ for the proposed MBMT.

Event mutants of an ESG can be generated by using the *event insertion* operator (*eI*-operator) to insert an event $e \notin E$ into the ESG resulting in $E^* = E \cup \{e\}$ or the *event omission* operator (*eO*-operator) to delete an event $e \in E$ from the ESG resulting in $E^* = E \setminus \{e\}$. The *eI* and *eO*-operators are applied similarly to ESG as the corresponding *nI* and *nO*-operators to DG. For the *eI*-operator, similar to the *nI*-operator, there is a need to insert an additional incoming/outgoing arc to and from the inserted event. Accordingly, for the *eO*-operator there is a need to remove the arcs connected with the deleted event.

Sequence mutants of an ESG can be generated by using the *sequence insertion* operator (*sI*-operator) to insert an arc (which is not in *A*) into the ESG or the *sequence omission* operator (*sO*-operator) to delete an arc (which is in *A*) from the ESG. The *sI* and *sO*-operators are applied similarly to ESG as the corresponding *aI* and *aO*-operators to DG. After applying the *sO* or *sI*-operators to an ESG, the validity of the resulting ESG has to be checked. We also have the *event corruption* operator (*eC*-operator) and the *sequence corruption* operator (*sC*-operator) applied similarly to ESG as the corresponding *nC* and *aC*-operators to DG (see Section 4.1).

The *eI*, *eO*, *sI*, *sO*, *eC* and *sC*-operators, and their combinations, can be applied repeatedly to an ESG multiple times.

4.3 SC-Based Mutation

If the underlying model in the proposed MBMT approach is an SC, it is called an SC-based mutation. *State mutants* of an FSM or SC can be generated by using the *state insertion* operator (*stI*-operator) or the *state omission* operator (*stO*-operator). The *stI*-operator inserts a state (which is not currently in the statechart) into *S*. It also requires the insertion of an additional incoming/outgoing transition-event pair to/from the inserted state. The *stO*-operator deletes an existing state from *S*. All transition-event pairs connected with the deleted state should also be removed from the FSM. Similarly, *transition mutants* of an FSM or SC can be generated by using the *transition insertion* operator (*tI*-operator) or the *transition omission* operator (*tO*-operator).The *tI*-operator inserts a transition $tr=(s, s')$ into *TR*, where s and $s' \in S$, but $(s, s') \notin TR$. The *tO*-operator deletes an existing transition *tr* from *TR*. The *stI*, *stO*, *tI* and *tO*-operators are done in a similar way as the *nI*, *nO*, *aI* and *aO*-operators for DG, respectively. After the application of the *stI*, *stO*, *tI* and *tO*-operators, the function *f* needs to be updated accordingly to add or delete appropriate mappings between transitions and events.

We also have the *state corruption* operator (*stC*-operator) and the *transition corruption* operator (*tC*-operator) for FSM and SC, which work the same way as the *nC* and a*C*-operators for DG. Finally, the *stI*, *stO*, *tI*, *tO*, *stC* and *tC*-operators, and their combinations, can be applied repeatedly to an FSM or SC multiple times.

4.4 Elementary Mutation Operators Compared With Those From Literature

TABLE 1 summarizes the differences between the code-based mutation operators reported in the literature and the elementary mutation operators proposed in this paper.

TABLE 1. Comparison between code-based mutation operators and the proposed elementary operators

	Code -based mutation operators	**Elementary mutation operators**
Number of operators	typically a small to medium-sized set of operators (e.g., [29]); some studies use a large set of operators (e.g.,[5])	two (insertion and omission)
Fault model	operators are chosen to simulate faults introduced by common programming mistakes	no explicit fault models
Construction	by experience and empirical results	systematically
Order of mutants typically used	first-order mutants	first-order mutants; second-order mutants (corruption operators)
Reducibility	code-based operators can be derived from elementary operators and their combinations	atomic; cannot be further reduced to other operators

For discussion purposes, we use the mutation operators reported by Fabbri et al. [10,12] as the example. These operators generate mutants for FSM and SC. The first two columns of TABLE 2 give operators defined in [10] and [12], whereas the last three columns show how they can be replaced by using an insertion or omission operator or a combination of these two. For example, referring to the first row, there is an operator in [10] and [12] to generate mutants with "wrong-start-state". These mutants can also be generated by applying either an insertion or an omission operator to the "start-state". Another example is the operator to generate the "event-exchanged" mutants in [10] and [12] can be replaced by an omission operator followed by an insertion operator on the related events.

There are two advantages of using the proposed elementary mutation operators over those suggested by Fabbri et al. First, the number of operators can be reduced to make mutation testing easier to understand. Second, a systematic application of the elementary operators to FSM and SC can help testers avoid possible inconsistencies of certain operators being applied to FSM but not to SC, or vice versa. It has been noted that the operator to generate mutants by deleting one of the existing states is included in [12], but not in [10] (see the second row from the bottom in TABLE 1), and the operator to generate mutants by inserting an additional state is included in [10], but not in [12] (see the last row). Such inconsistency is counter-intuitive as one would assume if mutants with a missing state are generated, then mutants with an additional state would also be generated. There is no explanation in [10] and [12] as to why such inconsistency occurs. However, a systematic application of the elementary operators can help testers easily discover this kind of inconsistency. The same information can also be used to identify what kind of additional mutants may be generated.

TABLE 2. Mutation operators in [10] and [12] and their equivalence using the elementary operators

Mutation operators in [12] (first column) and [10] (second column)		insertion	omission	artifact
wrong-start-state	wrong-starting-state (default state)	×	×	start-state
arc-missing	arc-missing		×	arc
event-missing	event-missing		×	event
event-extra	event-extra	×		event
event-exchanged	event-exchanged	×	×	event
destination-exchanged	-	×	×	destination
output-missing	output-missing		×	output
output-exchanged	output-exchanged	×	×	output
-	output-extra	×		output
state-missing	-		×	state
-	state-extra	×		state

5 Case Studies

Three case studies were conducted using industrial and commercial real-life systems: an interactive commercial music management system, a reactive/adaptive, real-time cruise control system, and a proactive control display unit of a marginal strip mower mounted on a truck. The first two studies use ESG (see Definition 2) to model the system behavior, whereas the third study uses ESG and SC (see Definition 5). Objectives of these studies are:

- Evaluate the fault detection capability of test sets generated using the proposed MBMT approach as described in Fig. 5 with ESG and SC as the models which is done by
  - determining whether there are still remaining faults in SUC that can be detected by such test sets
  - measuring the scores *MFDS* (see Definition 12) and MM*S* (see Definition 13)

- Compare the fault detection capability of test sets generated by the ESG-based mutation versus the SC-based mutation
  - test sets generated based on the event and sequence mutants of an ESG
  - test sets generated based on the state and transition mutants of an SC

For ESG-based mutation, first-order mutants were generated by applying the *sI*, *sO*, and *eO*-operators exactly once to the ESG. The process in Fig. 2 was used to generate a test set for these mutants covering all the event sequences of length 2 (i.e., covering all the events and arcs in an ESG). Similarly, for SC-based mutation, first-order mutants were generated by applying the *tI*, *tO*, and *stO*-operators exactly once to the SC. The process in Fig. 3 was used to generate test sets for these mutants covering all transition sequences of length 1 (i.e., covering all the states and transitions in an SC). Note that the *eI* and *stI*-operators would generate a large number of mutants. Moreover, these mutants represent a similar fault model as *sI* and *tI*-mutants in the context of graphical user interfaces. Hence, for reasons of cost and practicability, *eI* and *stI*-mutants were excluded in the case studies. As a result, the number of test sets for the ESG-mutation is the number of valid first-order mutants generated by the *eO*, *sI* and *sO*-operators, whereas the number of test sets for the SC-mutation is the number of valid first-order mutants generated by the *stO*, *tI* and *tO*-operators. However, we emphasize that the cost of test execution primarily depends on the size (in terms of the number of events) of all the test cases in each set rather than the number of test sets itself. This size varies for each ESG or SC and is determined by the algorithms used to solve the Chinese Postman Problem with respect to the underlying model. As explained in Fig. 2 and Fig. 3, the total size of each set is minimized in order to reduce the cost of test execution. Also note that MBT was conducted beforehand based on ESG and SC using the test generation processes described in Fig. 2 (with n=2) and Fig. 3 (with n=1). All the faults detected in SUC by MBMT were not found by MBT.

5.1 Description of the SUC

Real Jukebox (RJB) is an interactive personal music management program developed by Real Networks [30] for PCs running the Microsoft Windows operating system. The basic English version of Real Jukebox 2 (build 1.0.2.340) of Real Networks was used as the SUC for the first case study. The main GUI of RJB (see Fig. 10) has several pull-down menus that invoke other components. In this study, RJB was installed on a PC running Windows XP with Service Pack 2 satisfying the recommended hardware requirements (300 MHz CPU, 64 MB RAM, full duplex sound card, 16 bit color video card) as described in the product's manual. Due to the lack of a manufacturer's system specification, that is, a functional description of RJB, the *Help* contents and handbook of RJB were used to produce ESG for generating test cases (each of which is a CES).

The second case study comes from the automotive industry. It is a driver assistance system manufactured by Hella Corp. [31], one of the major German suppliers for electronic devices. The focus is on the adaptive cruise control (ACC) – an electronic control unit (ECU), which is an embedded reactive system (see Fig. 11). To automatically control the distance between vehicles, ACC is supplemented with a 24 GHz radar and connected over a CAN bus (controller area network) to five other ECU and thereby indirectly coupled to more than 7 sensors. Appropriate events were selected for constructing ESG to model the SUC that are specific to the system and test environment, for example, short circuits. Arcs connect these events according to the system specification and logical relations. For execution of the tests, a company-developed test environment was used that included a PC, SUC, a PLC (programmable logic controller), a programmable power supply, and the standard software CANoe [32] which provides simulation and monitoring of specific traffic scenarios as well as simulation of all other involved ECU and creation of test reports based on XML and HTML. While generating the test sequences for the software, CANoe state-based information was augmented by conditions and variables.

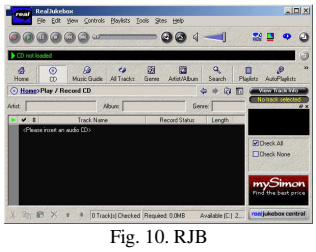

Fig. 10. RJB

Fig. 11. ACC

Fig. 12. RSM13

The third case study is ~~on~~ a proactive system to control a marginal strip mower (RSM13, see Fig. 12) mounted on the Unimog, a truck manufactured by Mercedes-Benz [33]. Considering safety aspects, the most significant component of the strip mower are revolving knives. Operation is affected either by the power hydraulic of the light truck or by the front power take-off. The position of the mow head changes continuously. An alteration from working on the left side to working on the right side is also possible. Besides the positioning, the incline and support pressure of the mowing unit can also be controlled. The control terminal has four different modes. The first mode is used to control the actuators, the second mode is used to switch between transport and working position, the third and the fourth are two different operation modes.

These four operation modes have been modeled by ESG and SC, respectively. An ESG and an SC for the second mode are shown in **Fehler! Verweisquelle konnte nicht gefunden werden.** and
Fig. 14, respectively. Refer to http://quebec.upb.de/MBMT.pdf for more information. Note that these two models were constructed independently for comparing test sets generated using the ESG-based mutation against those using the SC-based mutation, with respect to their fault detection capabilities.

All three SUTs are real world products where the source code was unavailable due to being a commercial-off-the-shelf product. As MBMT is a black-box testing technique, this work did not need to have the source code available. Since test case generation is based on models, test data has been included in the models themselves, wherever appropriate. More precisely, these algorithms are based on transitions or nodes randomly selected from the models, the amount of input data depending on the number of test cases generated. For RJB, manual test execution is performed on a windows client. Test cases are manually encoded for ACC and subsequently executed automatically. For RSM13, test cases are manually executed via control terminal.

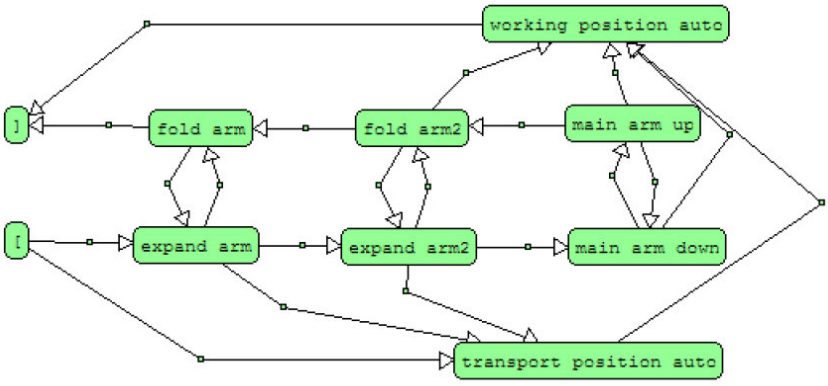

Fig. 13. An ESG to model the switch of RSM13 between its transport and working positions

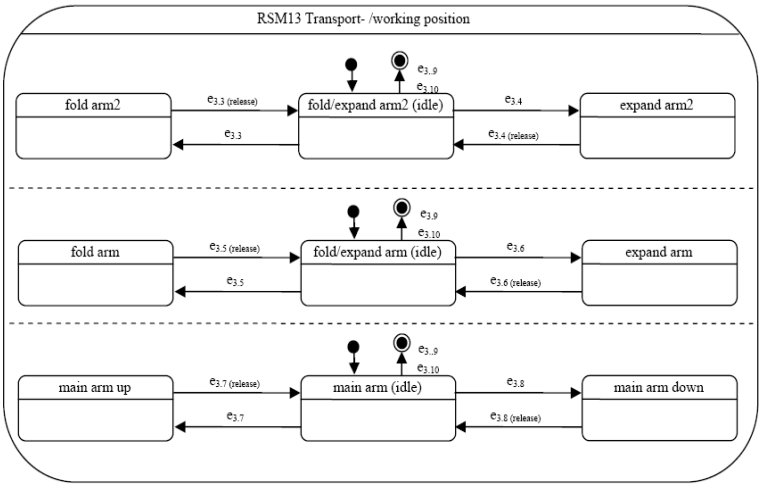

Fig. 14. A statechart to model the switch of RSM13 between its transport and working positions

5.2 Results, Analysis and Discussion

Having established a set of mutation operators, one question of interest is whether mutants generated with respect to a particular operator lead to a more effective detection of faults in the SUC. Furthermore, are some types of the generated mutants easier to kill than others? The data that follows aims to address these questions.

Mutants generated for RJB, ACC and RSM13 are classified based on the scheme in Fig. 6. TABLES 3, 4 and 5 give the number of *live* (LM_{ne1}, LM_{ne2}), *killed* (KM_a, KM_{na}) and *equivalent* (LM_e) mutants with respect to each mutation operator.

Also included are the mutation fault detection score (*MFDS*) and the modified mutation score (*MMS*). For example, referring to the second row of
TABLE 5, there are 22 *sO*-mutants (of which 5 are classified as LM_{ne2}, 16 as KM_a and 1 as KM_{na}) and 26 *tO*-mutants (of which 8 are classified as LM_{ne2}, 17 as KM_a and 1 as KM_{na}). The *MFDS* and *MMS* for the *sO*-mutants are 0.05 and 0.73, respectively, and 0.04 and 0.65 for the *tO*-mutants.

TABLE 3. Classification of ESG mutants generated for RJB, and the corresponding MFDS and MMS[4]

	LM_{ne1}	LM_{ne2}	LM_e	KM_a	KM_{na}	Sum_1	*MFDS*	*MMS*
sI	0	0	0	182	18	200	0.09	0.91
sO	0	142	0	55	3	200	0.02	0.28
eO	0	59	0	118	1	178	0.01	0.66
Sum_2	0	201	0	355	22	578	0.04	0.61

TABLE 4. Classification of ESG mutants generated for ACC, and the corresponding MFDS and MMS

	LM_{ne1}	LM_{ne2}	LM_e	KM_a	KM_{na}	Sum_1	*MFDS*	*MMS*
sI	0	21	0	57	4	82	0.05	0.70
sO	0	46	0	15	1	62	0.02	0.24
eO	0	11	0	25	0	36	0.0	0.69
Sum_2	0	78	0	97	5	180	0.03	0.54

TABLE 5. Classification of ESG and SC mutants generated for RSM13, and the corresponding MFDS and MMS

	LM_{ne1}	LM_{ne2}	LM_e	KM_a	KM_{na}	Sum_1	*MFDS*	*MMS*
sI / *tI*	0 / 0	0 / 0	0 / 0	179 / 36	5 / 5	184 / 41	0.03 / 0.12	0.97 / 0.88
sO / *tO*	0 / 0	5 / 8	0 / 0	16 / 17	1 / 1	22 / 26	0.05 / 0.04	0.73 / 0.65
eO / *stO*	0 / 0	3 / 3	0 / 0	12 / 12	0 / 0	15 / 15	0.0 / 0.0	0.80 / 0.80
Sum_2	0 / 0	8 / 11	0 / 0	207 / 65	6 / 6	221 / 82	0.03 / 0.07	0.94 / 0.79

TABLES 6, 7 and 8 give a brief description of each fault in SUC detected by the proposed MBMT approach, but which were not discovered before public release. To save space, only an excerpt of the corresponding faults is presented here; for a complete list refer to http://quebec.upb.de/MBMT.pdf. The fact that our approach could detect 6 faults in a released version of RSM13 (the SUC used in the third case study) that were not previously found during the testing conducted by a technical control board in Germany is especially important. Two of these faults are extremely safety-critical. Fault 1 indicates that the cutting unit can be activated without having any pressure on the bottom, which is very dangerous if pedestrians approach the working area. Fault 6 is observed by keeping the button for shifting the mow head pressed and changing to another screen. The mower head with its cutting unit cannot immediately be stopped in an emergency. Furthermore, restarting the hydraulic gear while it is already running can cause serious damage.

The last column of TABLES 6, 7 and 8 indicate which type of mutants the fault revealing test sets (the test sets that detected the remaining faults in the SUC) were generated. For example, of the 22 faults detected in RJB, 18 fault revealing test sets were generated with respect to the *sI*-mutants, 3 test sets with respect to the *sO*-mutants, and 1 test set with respect to the *eO*-mutants. Of the 5 faults detected in ACC, 4 fault revealing test sets were generated with respect to the *sI*-mutants, 1 test set with respect to the *sO*-mutants, and none with respect to the *eO*-mutants. Of the 6 faults detected in RSM13, 5 fault revealing test sets were generated with respect to the *sI* or *tI*-mutants, 1 test set with respect to the *sO* or *tO*-mutants, and none with respect to the *eO* or *stO*-mutants. With respect to our data, it is clear that test sets generated with respect to the *sI*-mutants are more effective in detecting faults in SUC than those generated with respect to the *sO*-mutants, which in turn are more effective than those generated with respect to the *eO*-mutants. It is also observed that test sets generated using the SC-based mutation (with respect to the *tI*, *tO* and *stO*-mutants) have the same fault detection capability as test sets generated using the ESG-based mutation (with respect to the *sI*, *sO* and *eO*-mutants).

Comparing the data in the last column of TABLES 6, 7 and 8 with the data in the 5th column (marked as KM_{na}) of TABLES 3, 4 and 5, a very important point worth noting is that the number of fault revealing test sets with respect to each type of mutant is the same as the number of mutants classified as KM_{na} for each type. Also, the total number of faults in SUC detected by using MBMT is the sum of all the mutants classified as KM_{na} and LM_{ne1}. For example, there are 22 faults detected for RJB which is the sum of 18 (number of *sI*-mutants classified as KM_{na}) + 3 (number of *sO*-mutants classified as KM_{na}) +

[4]The notation Sum_1 gives the total number of mutants with respect to each mutation operator, whereas Sum_2 gives the total number of mutants with respect to each classification as presented in Fig. 6. The same applies to Tables 4 and 5.

1 (number of *eO*-mutants classified as KM_{na}) + 0 (number of any mutants classified as LM_{ne1}). This is also the reason why the mutant fault detection score MFDS is defined as the ratio of ($|LM_{ne1}| + |KM_{na}|$) to ($|\mathcal{M}^*| - |LM_e|$) (see Definition 12).

TABLE 6. Description of 5 (of the 22) faults detected in RJB by the proposed MBMT approach

Fault ID	Description of the fault	Operator
1	*Track Position* cannot be set anymore and the application freezes when the end of track is reached.	*sI*
2	GUI shows state *Paused* but the music is still playing.	*sI*
3	Pushing P*lay* button plays a wrong track.	*sI*
4	*AutoPlay* does not start playing when a CD is inserted.	*sI*
5	*Pause* button has no impact despite the fact that the system is playing.	*sI*

TABLE 7. Description of the faults detected in ACC by the proposed MBMT approach

Fault ID	Description of the fault	Operator
1	The system resides in the error state and the speed of the vehicle drops to 5 km/h. If a simulation is now requested, the system remains in the error state (instead of switching to the simulation state as required).	*sO*
2	If an error occurs, the system does not switch in time to the error state.	*sI*
3	If the system resides in the error state and the error vanishes, the system does not switch to the requested active state in time.	*sI*
4	If the system resides in the error state and the error vanishes, the system does not switch to the requested inactive state in time.	*sI*
5	The system does not switch from init state to active/inactive state in time.	*sI*

TABLE 8. Description of the faults detected in RSM13 by the proposed MBMT approach

Fault ID	Description of the fault	Operator
1	When the function *bearingPressure* is deactivated, the function *cuttingUnit* can be activated.	*sI* for ESG or *tI* for SC
2	When the function *cuttingUnit* is activated, the function *bearingPressure* can be deactivated.	*sI* for ESG or*tI* for SC
3	When Engine I is activated, a change from the view *RSM_Actuator* to the view *Start* is possible.	*sI* for ESG or *tI* for SC
4	When Engine II is activated, a change from the view *RSM_Actuator* to the view *Start* is possible.	*sI* for ESG or *tI* for SC
5	When Axis Lock is activated, a change from the view *RSM_Actuator* to the view *Start* is possible.	*sI* for ESG or *tI* for SC
6	A change from the view *RSM_Operation_II* to the view *RSM_Operation_I* while function *ArmShiftLeft* is active is possible if the function *ArmShiftLeft* is activated.	*sO* for ESG or *tO* for SC

In all cases, the sets LM_{ne1} and LM_e (equivalent mutants) are empty. We also observe that the majority of the *sI*-mutants (182 of 200 for RJB, 57 of 82 for ACC, and 179 of 184 for RSM13) are classified as KM_a. The reason is that inserting an additional sequence (especially those related to the GUIs of RJB and RSM13) will most likely introduce a non-executable sequence of events, which can be easily detected. One example is a button that should be disabled in a certain context but becomes enabled due to this additional sequence (arc). Consequently, the modified mutation scores (MMS) for *sI*-mutants are high (0.91 for RJB, 0.70 for ACC, and 0.97 for RSM13). The same applies to the *tI*-mutants.

In contrast, a good number of mutants generated by the *sO*-operator are still live (refer to the set LM_{ne2}; 142 of 200 for RJB, 46 of 62 for ACC, and 5 of 22 for RSM13). This can be explained by the fact that when a sequence is deleted from the underlying ESG during the mutant generation, it in general represents certain features that should be included but are missing from the mutant because of this deletion. However, it is more difficult detecting such a fault as testers may not be aware of what is missing. As a result, the corresponding modified mutation scores (MMS) for *sO*-mutants (0.28 for RJB, 0.24 for ACC, and 0.73 for RSM13) are also relatively lower than those for the *sI*-mutants. The same also applies to the *tO*, *eO* and *stO*-operators.

TABLE 9 compares all three case studies with respect to the ESG-based *sI*, *sO* and *eO*-mutants. The percentage (α) corresponds to the ratio of the number of mutants of a given type to the total number of generated mutants. For example, in the case of the *sI*-mutants, α equals 34.6% for RJB as 200 of the 578 mutants are generated by the *sI*-operator. Similarly, the percentage (β) corresponds to the ratio of the number of faults in SUC being detected by test sets generated based on a given type of mutants to the total number of faults detected in SUC. For example, in the case of the *sI*-mutants, β equals 81.8% for RJB as 18 of the 22 faults are detected by test sets generated based on the *sI*-mutants. The percentage (γ) corresponds to the ratio of the number of faults in SUC being detected by test sets generated based on a given type of mutants to the number of mutants of that type. Overall, our data suggests that not only are test sets generated with respect to the *sI* and *tI*-mutants more likely to detect faults in SUC (as discussed above) but also these mutants themselves are easier to kill than *sO*, *tO*, *eO*, and *stO* mutants.

TABLE 9. Comparison of ESG-based mutants

	RJB			ACC			ESM13		
	α	β	γ	α	β	γ	α	β	γ
sI	34.6	81.8	9 (18/200)	45.5	80 (4/5)	4.8 (4/82)	83.2	83.3 (5/6)	2.7 (5/184)

	(200/578)	(18/22)		(82/180)			(184/221)		
sO	34.6 (200/578)	13.6 (3/22)	1.5 (3/200)	34.4 (62/180)	20 (1/5)	1.6 (1/62)	9.9 (22/221)	16.6 (1/6)	4.5 (1/22)
eO	30.8 (178/578)	4.5 (1/22)	0.6 (1/178)	20.0 (36/180)	0 (0/5)	0 (0/36)	6.8 (15/221)	0 (0/6)	0 (0/15)
Sum	578	22		180	5		221	6	

The time for model creation can be neglected as this is primarily a product of MBT, which is in general assumed as the preceeding step to MBMT. The effort for test generation has also a minor part in time cost as this has been accomplished by tools. Therefore, the main part of cost is the total time of test execution, which is shown in Table 10. The number of test cases also does not say much about test effort without treating test sequence length. In our case studies, we have used algorithms to minimize the overall length – not the number of test cases. Typically, test case generation results only in a few test cases for each mutant – in best case even only one. For details, refer to [7][12]. Average time spent to execute test cases manually for each mutant is 3,11 minutes for RJB, whereas it took 4,75 minutes in average for RSM13. Since test execution for ACC is performed in a different way, i.e. by first encoding test cases manually and then executing automatically, average time spent to execute test cases for each mutant is 20 minutes.

TABLE 10. Average time spent for test execution

SUC	Number of mutants	man-days (student tester, 6h/day)	hours in total	average time (in minutes) for each mutant	Comments
RJB	578	5	30	3,11	Manual test execution on windows client
ACC	180	10	60	20,00	Manual encoding of test cases and subsequent automatic execution
RSM13 (ESG+SC)	303	4	24	4,75	Manual test execution via control terminal
Sum	1061	19	114	6,45	

5.3 Tool Support

The approach described in the previous sections is straightforward with a simple structure. The large amount of data to be gathered and analyzed is, however, hardly feasible and, equally critical, error-prone when processed manually. Therefore, a chain of tools, available under a uniform graphical user interface (GUI), was developed to cope with the scalability problem in large projects. They have also been used to conduct the case studies in Section 5. Some of the tools are add-ons to commercial ones, while others implement the test algorithms described in the previous sections. An ESG-based *mutant and test set generator* (MTSG) forms the heart of this tool chain. A snapshot in Fig. 15 represents its entry window. For mutant generation, MTSG computes first-order mutants with respect to each operator. For example, the configuration in Fig. 16 indicates that 50% of the *sI*-mutants, 150 of the *sO*-mutants, 0% of the *eI*-mutants, and 100% of the *eO*-mutants are generated.

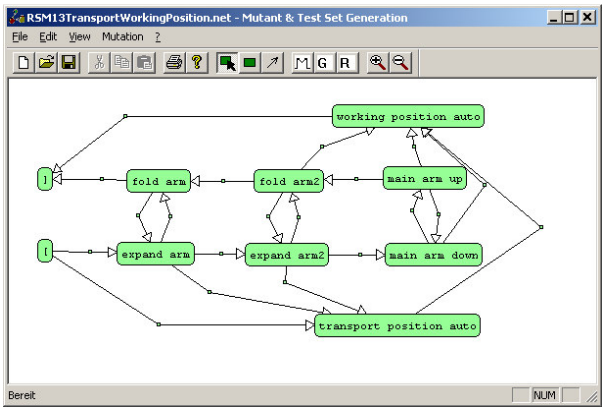

Fig. 15. The entry window of MTSG

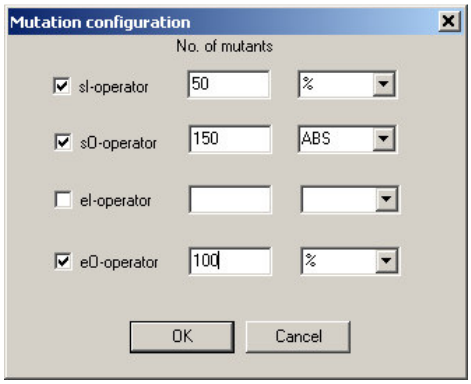

Fig. 166. Window for specifying the number or percentage of mutants to be generated with respect to each operator

For ESG-based test generation, MTSG checks the validity of the model analyzed as the first step before a test set is generated to cover all event sequences of length *n*. Each test case in this set is a CES (a complete event sequence) of the ESG.

To reduce the execution cost, another unit of MTSG generates a test set in such a way that the total length of all the CES contained is minimal. This is done by using the algorithm for solving the Chinese Postman Problem.

5.4 Threats to Validity

There may be several threats to the validity of this paper that include, but are not limited to, the following.

A threat to the construct validity is the assumption that the *competent programmer hypothesis* [26,27] and the *coupling effect* [28] can also be applied to MBMT. Nevertheless, we point out that mutants generated by the *corruption* operators (such as the *eC* and *sC*-operators in Section 4) in our approach are regarded as second-order mutants because such corruption can be viewed as an *omission* operation followed by an *insertion* operation. However, the same mutants are classified as first-order mutants in many other studies as they are considered to be generated by the application of a *single* mutation operator (even though that operator can be further decomposed into two elementary operators – *omission* and *insertion* – as discussed above). This equivalence between first-order mutants in other studies and second-order mutants in our approach implies that the coupling effect for some first-order mutants in those studies can be equivalently applied to the related second-order mutants in our study.

As for the internal validity, one concern is the types of mutation operators used and the number of mutants generated by these operators. Obviously, the more mutants we use, the higher the probability that test sets generated with respect to these mutants (following the process in Fig. 2 and Fig. 3) can help us detect faults in SUC. Also, one may argue that in addition to the first-order mutants, higher-order mutants should be used as well because they might be more appropriate to simulate real faults [34,35]. However, the disadvantage is that this will also increase the number of mutants and the cost of mutation testing significantly. Thus, in order to strike a balance between the cost-effectiveness of mutation testing and the fault detection capability (in terms of revealing faults in SUC) of test sets generated with respect to the mutants, it is a necessary step to limit the number of mutation operators and the number and order of the mutants generated.

Another concern is that only event sequence graphs (ESG) and statecharts (SC) were used for modeling SUC. However, such choice is based on their intensive use of formal graph-theoretical notions and algorithms. With respect to the algorithm (Φ) used for generating test sets based on a mutated model ($\mathcal{M}^*$), only sequence-based coverage criteria are considered. More precisely, the ESG-based test generation process described in Fig. 2 was used to generate sets of complete event sequences (CES) to cover all event sequences of length 2, and the SC-based test generation process described in Fig. 3 was used to generate sets of complete transition sequences (CTS) to cover all transition sequences of length 1. One would argue that additional coverage criteria could also be used for test generation. While this is true, our decision was once again based on the cost-effectiveness and the fault detection capability as discussed above.

Regarding external validity, since our evaluation of the proposed MBMT approach has primarily been based on empirical data, arguably our results may not be generalized to all systems. However, the three systems used in our studies (Real Jukebox – a personal music management program, ACC – the adaptive cruise control of a driver assistance system, and RSM13 – the control terminal of a marginal strip mower) vary greatly from the other in terms of the functionality, complexity, operational parameters and environments, etc., and this gives us confidence in the general applicability of MBMT to different systems.

6 Related Work

Mutation testing was originally proposed as a white-box, fault injection-based technique for implementation-based software testing, especially at the unit-level. Recent works have also applied it to other testing levels such as integration-level, and other artifacts such as program specifications. An overview of variations of mutation testing related to the one proposed in this paper is presented focusing on the mutation operators used to generate these mutants. Interested readers may also refer to a survey on mutation testing for more details [25].

6.1 Code-Based Mutation Testing

In implementation-based mutation testing, mutants are generated by applying mutation operators to the source code to introduce simple syntactic changes. These operators are language dependent. For example, we have different sets of operators for Fortran 77 [6], C [5], COBOL [36], Lisp [37], Ada [38,39], and Java [40,41,42]. The operators can be further di-

vided. *Code-based mutation operators* are the operators developed for testing a function or a unit of procedural programs, such as the 22 operators used by Mothra [29] for Fortran programs and the 77 operators used by Proteum [5,43] for C programs. *Interface mutation operators* include operators such as those used by Proteum/IM [7] that apply mutation testing at the integration-level. Test cases generated to kill these mutants provide a good indication of how well the interactions between different modules of an application have been tested. The focus is on the source code related to module interactions, including function calls, parameters, global variables, etc. As the object-oriented (OO) paradigm becomes more popular, a new category of mutation operators, such as the class and inter-class mutation operators [40,41,42], is needed to handle inheritance, polymorphism, dynamic binding, and other OO specific features.

6.2 Model-Based Mutation Testing

In addition to source code, mutation operators can also be applied to other artifacts including program specifications, as well as architecture and design models. For each mutation testing method, it can be further divided by whether it is for testing the specification/model itself or for testing the corresponding implementation based on the specification/model. Studies such as [15] follow the procedure in Fig. 4 to test the specification, whereas studies such as [44] test the implementation. More precisely, Ammann et al. [44] use model checkers to generate test cases based on the model checker specifications. Such tests have the property to distinguish the original and mutated models. These tests are then used to validate the corresponding implementation. In [44], selection of mutation operators is based on experience and it is claimed that they have reasonable coverage. However, the proposed method in this paper uses two simple mutation operators to systematically and automatically generate model mutants and then generate test cases. In the literature, there is no proposed systematic approach for generation of fault models from mutation operators but instead experience-based approaches. [44] presents an experience-based approach.

For mutation testing methods that follow the procedure described in Fig. 4, test cases are executed on the original specification/model and the mutated specifications/models (namely, the mutants) to see how many mutants they kill. This type of mutation testing has some benefits over the implementation-based mutation testing discussed in Section 6.1. For instance, the former may help programmers detect a missing path error [45]; that is, a situation when an implementation neglects an aspect of a problem, and a section of code is altogether absent. Since there is no evidence in the code itself for the omission, such errors are hard to find by analyzing the code alone. Furthermore, implementation-based testing might not be feasible for some systems because testers do not have access to their source code. Generally, generating tests from a specification can proceed independently of program development.

An overview of studies that have proposed different sets of mutation operators for various specifications and models is presented in the following.

Kovács et al. [13] have used mutation testing for the conformance testing of telecommunication protocols specified using SDL. SDL is a language for the specification and description of telecommunications systems. Their focus is the automation of the test generation and selection process using mutation testing. For this purpose, they use six types of mutation operators: state modification, input modification, output modification, action modification, predicate modification, and save missing operators. Sugeta, Maldonado, and Wong [15] proposed the use of mutation testing to test specifications written in SDL that is different from the approach of Kovács et al. [13] because it focuses on testing the specification itself as opposed to the implementation. Considering intrinsic features of SDL, they proposed three classes of mutation operators: process mutation, interface mutation, and structure mutation.

Fabbri et al. [10] defined a set of mutation operators to apply mutation testing in the context of FSMs. These operators are based on the error classes defined by Chow [9] and on the results of their heuristic research [10]. Nine operators were developed; namely wrong-start-state, arc-missing, event-missing, event-extra, event-exchanged, destination-exchanged, output-missing, output-exchanged, and state-missing. Later, they developed Proteum/FSM [46] to support their approach. Fabbri et al. [11] also investigated mutation testing on Petri nets. They designed a preliminary set of mutation operators for Petri nets: input-missing, input-extra, input-shifted, input-exchanged, output-missing, output-extra, output-shifted, output-exchange, and wrong-initial-marking. Fabbri et al. [12] developed mutation operators for statecharts as well. The operators are divided into three categories: finite-state machine (FSM) mutation operators, extended FSM (EFSM) mutation operators, and statecharts-feature-based operators. El-Fakih et al. [47] proposed an FSM based incremental and mutation testing approach for the implemantations that have more states than their specification, where user defined faults are implemented in an arbitrary way.

Clark et al. [48] described semantic mutation testing, which mutates the semantics of the language in which the description is written rather than mutating the description. Since mutations on the semantics of the language may cause possible misunderstandings of the description language, a different class of faults are created. They demonstrated their approach with statecharts and C code. Aichernig et al. [49] proposed a test case generation methodology supporting a wide range of UML constructs. Their methodology is grounded on the formal semantics of Back's action systems and input–output conformance relation. Their algorithms for test case generation work on the level of labelled transition system (LTS), the UML state machine model defined by the user has to be converted into an LTS via action systems. Their methodology employes mutation operators on the level of the specification to insert faults and generates test cases that will reveal the faults inserted. In another research, Aichernig and Jöbstl [50] developed an approach for refinement checking of action systems as a first step for test case generation from mutated action systems.

Object-Z [51] is an extension of Z to facilitate specification in an object-oriented style. Object-Z preserves all Z's syntax structures and semantics. Therefore, a Z specification is also an Object-Z specification. Object-Z does, however, extend Z with the object-oriented notions, for example, encapsulation, inheritance, and polymorphism, which are embodied in the class structure in Object-Z. The ideas used in implementation-based mutation take the types, variable names, and operators as the main source for mutants. Stocks [52] pointed out that these ideas can also be applied to Z specification and presented the mutants for the logic, set, and relation operators. Liu and Miao [53] named the mutation operators described by Stocks [52] as *traditional* mutation operators and proposed a set of additional operators. They presented five categories of mutation operators for an Object-Z specification: traditional mutation operators, Z-specific features, visibility, inheritance, and polymorphism. The first two categories are designed for a Z specification. The last three categories are designed for object-oriented features in an Object-Z specification.

Woodward [54] proposed OBJETEST, a prototype testing system for algebraic specifications. The algebraic specification language considered in the study is a version of OBJ implemented at UMIST [55] and subsequently marketed under the name ObjEx. Woodward used five types of mutation operators: alteration of operator attributes, replacement of non-constant operators in equations, replacement of constant operators in equations, replacement of VAR symbols in equations, and large-scale equation mutation.

Lee and Offutt [56] proposed a technique for using mutation testing to test the semantic correctness of XML-based component interaction. They introduced the Interaction Specification Model (ISM) to specify web software interaction. Two initial mutation operator classes were also proposed. Xu et al. [57] introduced a similar but more sophisticated approach. This approach is mainly different from [56] in that it mutates not XML directly but XML schema. A formal model for XML schema is also defined and a method to create a virtual schema when XML schemas do not exist is presented. Based on this formal model, they proposed mutation operators (called *perturbation* operators by Xu et al. [57]) that perturb the XML schemas. Test cases (as XML messages) were generated to kill the perturbed XML schema. Li and Miller [58] used the mutation technique to test semantics of W3C XML schema. It focuses on the flaws related to W3C XML schema special features, such as namespace, inheritance, element and attribute declaration, type facets, and so on. They presented a set of mutation operators to test for faults in the use of an XML schema.

Feature models used in software product lines describe the constraints that link the features to products and allow the configuration of tailored software products. A software product line containing hundreds of features may result in millions of possible software products. Therefore, testing a software product line is challenging due to the number of the possible products. Henard et al. [59] proposed two mutation operators to derive erroneous feature models (mutants) from an original feature model and assess the capability of the generated original test suite to kill the mutants. They showed that dissimilar tests suites have a higher mutant detection ability than similar ones.

Fuzz testing, or fuzzing in short, is a negative testing approach for robustness testing [60]. Some fuzzing approaches use models of protocols to generate test inputs [61, 62]. Set of test inputs can be extended by mutating the model under consideration. For instance, Zhang et al. [63] used a mutation-based approach for detecting implementation flaws of network protocol through fuzzing. Similarly, test inputs for fuzzing can be generated by mutating input grammars [64].

Ammann and Black used mutation testing and model checking for automatically producing tests from formal specifications [44] and measure test coverage on the specification itself [65]. To generate tests, the mutation operators are applied to all temporal logic clauses, resulting in a set of mutant clauses. The model checker compares the original state machine specification with the mutants. When an inconsistent clause is found, the model checker produces a counterexample if possible, and converts it into a test case. To measure the coverage of a test set, each test is turned into a finite-state machine that represents only the execution sequence of that test. Each state machine is compared by the model checker with the set of mu-

tants produced previously. Black et al. [66,67] refined the mutation operators defined by Ammann et al. [44] and proposed new ones to be applied in the same approach that combines mutation testing and model checking.

Test case generation using model checking is one of the test case generation techniques, such as an ESG-based test generation technique given in Fig. 2, and an FSM and SC-based test generation technique given in Fig. 3. Our mutant generation, execution and classification algorithm presented as Algorithm 1 uses any given test generation algorithm Φ, which could be also a test case generation technique using model checking. Therefore, it is generic, whereas the approach presented in [44] is specific. Moreover, our approach classifies mutants and investigates their properties, which is beyond the scope of [44].
In [44], two mutation operators M_1 and M_2 are used to introduce mutations to the state machine. M_1 "changes condition c_i by conjoining an additional condition $w = e$ for some variable w and some possible value e" [44]. M_2 "changes those conditions c_i which are multiple conjoined conditions by deleting one of the conditions" [44]. It is written that "other candidate mutant operators that we may use are changing the new value e_i to some other possible value, deletion of $c_i : e_i$; pairs, and replacing variables in some c_i with other variables of compatible type" [44]. It is indicated that "this list is clearly not exhaustive, but it does produce a set of mutants with reasonable coverage properties, as we document subsequently" [44]. However, "reasonable coverage" is not explained in a formal or systematic way. Moreover, why M_1 and M_2 are preferred over other candidate mutant operators listed in the paper is not explained.
Candidate mutant operators given in [44] do not include, for instance, addition of a new $c_i : e_i$; pair or negating one of the variables in some c_i. It is not explained why are they not included. Our observation is that in the literature there is no broadly accepted, systematic approach to fault model classification and generation, but instead experience-based approaches. [44] presents also an experience-based approach. The follow-on works of [44] by Belli and Guldali [68,69] extended the idea of using model checkers for test generation by generating a series of faulty models as inputs to a model checker to generate test cases, instead of solely one model.

The approach represented by Offutt et al. [70] thoroughly differs from the approach in the present paper for the following reasons explained in [71]. Offutt et al. [70] used grammar-based models and mutation operators, namely for duplication, deletion, and replacement of nonterminal and terminals of a given grammar. Belli and Beyazit [71] avoid the use of these operators for the following reasons:

- First, all of the operators, except nonterminal duplication, can be realized by using the combinations of the event-based operators.
- Second, and more critical reason, nonterminal duplication is not type-preserving. Consequently, if this operator is applied to an RG (Regular Grammar), the mutant can become a CFG (Context-Free Grammar); that is, the type of the original model is injured.
- Apart from likely improper manipulation of the functionality and expressive power of the original grammar, this kind of mutation has severe impacts on decidability features relative to the undecidability of the equivalence of generated CFG-type mutants.

In [71], this drawback is exemplified using the regular grammar in Fig. 8a that is drawn from an example used by Ammann and Offutt [72]. Belli and Beyazit have slightly, nevertheless equivalently, reformatted the example for saving space. The nonterminal duplication mutant in Fig. 8b is a CFG.

One of the major differences between all the mutation techniques discussed above and the MBMT proposed in this paper is that the former *empirically* determine a (possibly very large) set of mutation operators based on selected fault models, whereas the latter uses two elementary mutation operators – *insertion* and *omission*, and perhaps their combinations as well, if necessary. *Insertion* and *omission* mutation operators are also applicable to other models in addition to graph-like models. In [73], Belli et al. applied these two basic mutation operators to pushdown automata. Belli and Beyazit showed the use of these two basic mutation operators on regular expressions in [74] and also on regular grammars in [75].

7 Conclusions and Further Work

An innovative model-based mutation testing (MBMT) approach (see Fig. 5) has been proposed to generate mutants based on the *model* of the system under consideration (SUC), thereby injecting faults into the model rather than the implementation. In contrast to the code-based mutation testing (see Fig. 4), MBMT enables not only the application of mutation testing when source code of SUC is not available but also the evaluation of the fault detection capability (in terms of revealing faults in the SUC) of test sets generated based on the mutants. Another significant difference is that code-based mutation testing typically uses a small to medium-sized set of operators (e.g., 22 operators in Mothra [29]) with some studies using a much larger set (e.g., 77 operators for programs in C [5]), which often makes the mutation testing difficult to use. Our pro-

posed approach only uses two elementary mutation operators – *omission* and *insertion*. The syntax of these operators is explained by means of directed graphs (DG), which are then semantically enriched and exemplified using a collection of graph-based models (ESG, FSM, and SC). We have also demonstrated that these two elementary operators using systematic iterations and combinations can replace many mutation operators presented in the literature. The third difference is that mutants generated using MBMT should be classified into more categories as described in Fig. 6, rather than just *live*, *killed* and *equivalent* mutants as in the code-based mutation testing.

Three case studies were conducted to evaluate the applicability of the proposed MBMT approach on industrial and commercial real-life systems (namely, Real Jukebox, ACC and RSM13; see Section 5.1) and assess how likely that test sets generated to kill the ESG-based *eO*, *sI* and *sO*-mutants and the SC-based *stO*, *tI* and *tO*-mutants can also detect faults in SUC. Our experimental data shows that indeed MBMT can help testers detect faults in systems that were already released for public use. It is especially worth noting for the third case study as the system was tested by a German authority for testing and certification of technical systems, but the faults detected by our approach were not revealed. Also, the faults detected by MBMT in the first two case studies were not previously known to the manufacturers (i.e., were not detected by the manufacturers before they released the systems). Results from our case studies indicate that test sets targeting the *sI* and *tI*-mutants are not only more likely to detect faults in SUC (see TABLES 6, 7 and 8), but that the *sI* and *tI*-mutants are more easily killed than the *sO*, *tO*, *eO*, and *stO* mutants (see TABLES 3, 4 and 5). We conclude that, when testing resources are constrained, it is more efficient to focus on the *sI* and *tI*-mutants before turning to the *sO*, *tO*, *eO*, and *stO* mutants.

Our future work includes, but is not limited to, the use of different models such as General Net Theory (Petri nets), algebraic models like (extended) regular expressions, and generic rewriting systems like formal grammars, as well as various test generation algorithms to satisfy other criteria in addition to just covering all event sequences of length 2 for ESG and all transition sequences of length 1 for SC. Case studies using different types of mutants (including higher-order mutants) and more systems from different application domains are also underway.

References

[1] Object Management Group, Unified Modeling Language (UML), http://www.omg.org

[2] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, C. Willcock, An Introduction to the Testing and Test Control Notation (TTCN-3), The International Journal of Computer and Telecommunications Networking 42 (3) (2003) 375-403.

[3] F. Belli, C.J. Budnik, L. White, Event-based Modeling, Analysis and Testing of User Interactions: Approach and Case Study, Journal of Software Testing, Verification and Reliability 16 (1) 2006 3-32.

[4] F. Belli, A. Hollmann, Test Generation and Minimization with 'Basic' Statecharts, in: 23rd Annual ACM Symposium on Applied Computing, 2008, pp. 718-723.

[5] H. Agrawal, R.A. DeMillo, R. Hathaway, Wm. Hsu, W. Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur, E.H. Spafford, Design of Mutant Operators for the C Programming Language, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, W. Lafayette, Indiana, Mar. 1989.

[6] K.N. King, A. J. Offutt, A Fortran Language System for Mutation-based Software Testing, Software - Practice and Experience 21 (7) (1991) 685-718.

[7] M.E. Delamaro, J.C. Maldonado, A.P. Mathur, Interface Mutation: An Approach for Integration Testing, IEEE Transactions on Software Engineering 27 (3) (2001) 228-247.

[8] G.V. Bochmann, A. Petrenko, Protocol Testing: Review of Methods and Relevance for Software Testing, in: International Symposium on Software Testing and Analysis, 1994, pp. 109-124.

[9] T.S. Chow, Testing Software Design Modeled by Finite-state Machines, IEEE Transactions on Software Engineering 4 (3) (1978) 178-187.

[10] S.C.P.F. Fabbri, J.C. Maldonado, M.E. Delamaro, P.C. Masiero, Mutation Analysis Testing for Finite-state Machines, in: 5th IEEE International Symposium on Software Reliability Engineering, 1994, pp. 220-229.

[11] S.C.P.F. Fabbri, J.C. Maldonado, P.C. Masiero, M.E. Delamaro, W.E. Wong, Mutation Testing Applied to Validate Specifications Based on Petri nets, in: 8th International IFIP Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, FORTE 1995, 1995, pp. 329-337.

[12] S.C.P.F. Fabbri, J.C. Maldonado, T. Sugeta, P.C. Masiero, Mutation Testing Applied to Validate Specifications Based on Statecharts, in: 10th IEEE International Symposium on Software Reliability Engineering, 1999, pp. 210-219.

[13] G. Kovács, Z. Pap, D.L. Viet, A. Wu-Hen-Chang, G. Csopaki, Applying Mutation Analysis to SDL Specifications, SDL Forum, Stuttgart, Germany, 2003, pp. 269-284.

[14] B. Sarikaya, Conformance Testing: Architectures and Test Sequences, Computer Networks and ISDN Systems 17 (2) (1989) 111-126.

[15] T. Sugeta, J.C. Maldonado, W.E. Wong, Mutation Testing Applied to Validate SDL Specifications, in: 16th IFIP International Conference on Testing of Communicating Systems, Springer LNCS 2978 (2004) 193-208.

[16] W.E. Wong, A. Restrepo, Y. Qi, B. Choi, An EFSM-based Test Generation for Validation of SDL Specifications, in: 3rd International Workshop on Automation of Software Test, 2008, pp. 25-32.
[17] B.J. Choi, R.A. DeMillo, E.W. Krauser, R.J. Martin, A.P. Mathur, A.J. Offutt, H. Pan, E.H. Spafford, The Mothra tool set, in: 22nd Annual Hawaii International Conference on System Sciences, 1989, pp. 275-284.
[18] B.K. Aichernig, E. Joebstl, Efficient Refinement Checking for Model-Based Mutation Testing, in: 12th International Conference on Quality Software, Xi'an, China, Aug. 2012.
[19] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2000.
[20] H. Zhu, P.A.V. Hall, J.H.R. May, Software Unit Test Coverage and Adequacy, ACM Computing Surveys 29 (4) (1997) 366-427.
[21] A.M. Memon, An Event-flow Model of GUI-based Applications for Testing, Journal of Software Testing, Verification and Reliability 17 (3) (2007) 137–157.
[22] F. Belli, C.J. Budnik, Test Minimization for Human-computer Interaction, Journal of Applied Intelligence 26 (2) 2007 161-174.
[23] T.A. Budd, F.G. Sayward, R.A. DeMillo, R.J. Lipton, The Design of a Prototype Mutation System for Program Testing, in: National Computer Conference, 1978, pp. 623-627.
[24] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, IEEE Computer 11 (4) (1978) 34-41.
[25] Y. Jia, M. Harman, An Analysis and Survey of the Development of Mutation Testing, IEEE Transactions on Software Engineering 37 (5) (2011) 649-678.
[26] A.T. Acree, On Mutation, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.
[27] T.A. Budd, Mutation Analysis of Program Test Data, Ph.D. Thesis, Yale University, New Haven, Connecticut, 1980.
[28] A.J. Offutt, Investigations of the Software Testing Coupling Effect, ACM Transactions on Software Engineering and Methodology 1 (1) (1992) 3-18.
[29] R.A. DeMillo, R.J. Martin, The Mothra Software Testing Environment User's Manual, Technical Report SERC-TR-4-P, Software Engineering Research Center, Purdue University, W. Lafayette, Indiana, Sep. 1987.
[30] RealNetworks, Inc., http://www.real.com
[31] Hella KGaA Hueck & Co., http://www.hella.de
[32] Vector Informatik GmbH, http://www.vector-worldwide.com
[33] Mercedes-Benz, http://www.mercedes-benz.de
[34] M. Harman, Y. Jia, W.B. Langdon, A Manifesto for Higher Order Mutation Testing, in: 5th Workshop on Mutation Analysis, Paris, France, Apr. 2010.
[35] Y. Jia, M. Harman, Higher Order Mutation Testing, Journal of Information and Software Technology 51 (10) (2009) 1379-1393.
[36] J.M. Hanks, Testing COBOL Programs by Mutation, Technical Report GIT-ICS-80/04, Georgia Institute of Technology, 1980.
[37] T.A. Budd, R.J. Lipton, Proving Lisp Programs Using Test Data, Digest for the Workshop on Software Testing and Test Documentation, IEEE Computer Society Press, 1978, pp. 374-403.
[38] J.H. Bowser, Reference Manual for Ada Mutant Operators, Technical Report GIT-SERC-88/02, Georgia Institute of Technology, Feb. 1988.
[39] A.J. Offutt, J. Payne, J.M. Voas, Mutation Operators for Ada, Technical Report ISSE-TR-96-09, Department of Information and Software Systems Engineering, George Mason University, Fairfax Virginia, Mar. 1996.
[40] S.W. Kim, J.A. Clark, J.A. McDermid, Investigating the Effectiveness of Object-oriented Testing Strategies with the Mutation Method, Journal of Software Testing, Verification and Reliability 11 (4) (2001) 207-225.
[41] Y.S. Ma, Y.R. Kwon, A.J. Offutt, Inter-class Mutation Operators for Java, in: 13th IEEE International Symposium on Software Reliability Engineering, 2002, pp. 352-363.
[42] Y.S. Ma, A.J. Offutt, Y.R. Kwon, MuJava: An Automated Class Mutation System, Journal of Software Testing, Verification and Reliability 15 (2) (2005) 97-133.
[43] M.E. Delamaro, J.C. Maldonado, A.P. Mathur, Proteum – a Tool for the Assessment of Test Adequacy for C Programs: User's Guide, Technical Report SERC-TR-168-P, Software Engineering Research Center, Purdue University, W. Lafayette, Indiana, Apr. 1996.
[44] P. Ammann, P.E. Black, W. Majurski, Using Model Checking to Generate Tests from Specifications, in: 2nd IEEE International Conference on Formal Engineering Methods, ICFEM 1998, 1998, pp. 46-54.
[45] J.B. Goodenough, S.L. Gerhart, Toward a Theory of Test Data Selection, IEEE Transactions on Software Engineering 1 (2) (1975) 156-173.
[46] S.C.P.F. Fabbri, J.C. Maldonado, P.C. Masiero, M. Delamaro, Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing, in: 19th International Conference of the Chilean Computer Science Society, 1999, pp. 96-104.
[47] K.A. El-Fakih, R. Dorofeeva, N. V Yevtushenko, G. V Bochmann, FSM-based testing from user defined faults adapted to incremental and mutation testing, Programming and Computer Software 38 (4) (2012) 201–209.
[48] J.A. Clark, H. Dan, and R.M. Hierons, Semantic mutation testing, Science of Computer Programming 78 (4) (2013) 345–363.
[49] B.K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, S. Tiran, Killing strategies for model-based mutation testing, Journal of Software Testing, Verification and Reliability (2014) p. n/a–n/a.
[50] B.K. Aichernig, E. Jöbstl, Towards Symbolic Model-Based Mutation Testing: Combining Reachability and Refinement Checking, in: Workshop on Model Based Testing, 2012, pp. 88–102.
[51] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, Boston, 2000.
[52] P. A. Stocks, Applying Formal Methods to Software Testing, Ph.D. thesis, The University of Queensland, Brisbane, 1993.

[53] L. Liu, H. Miao, Mutation Operators for Object-Z Specification, in: 10th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2005, 2005, pp. 498-506.
[54] M. Woodward, Errors in Algebraic Specifications and an Experimental Mutation Testing Tool, Software Engineering Journal 8 (4) (1993) 211-224.
[55] D. Coleman, R.M. Gallimore, V. Stavridou, The Design of a Rewrite Rule Interpreter from Algebraic Specifications, Journal of Software Engineering 2 (4) (1987) 95-104.
[56] S.C. Lee, J. Offutt, Generating Test Cases for XML Based Web Component Interactions Using Mutation Analysis, in: 12th IEEE International Symposium on Software Reliability Engineering, 2001, pp. 200-209.
[57] W. Xu, A.J. Offutt, J. Luo, Testing Web Services by XML Perturbation, in: 16th IEEE International Symposium on Software Reliability Engineering, 2005, pp. 257-266.
[58] J.B. Li, J. Miller, Testing the Semantics of W3C XML Schema, in: 29th Annual International Computer Software and Applications Conference, COMPSAC 2005, 2005, pp. 26-28.
[59] C. Henard, M. Papadakis, G. Perrouin, J. Klein, Y. Le Traon, Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing, in: Sixth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013, 2013, pp. 188–197.
[60] P. Oehlert, Violating assumptions with fuzzing, IEEE Journal on Security & Privacy 3 (2) (2005) 58-62.
[61] Y. Hsu, G. Shu, and D. Lee, A model-based approach to security flaw detection of network protocol implementations, in: Proc. IEEE International Conference on Network Protocols (ICNP), pp. 114-123, 2008.
[62] S. Gorbunov and A. Rosenbloom, Autofuzz: Automated network protocol fuzzing framework, IJCSNS International Journal of Computer Science and Network Security 10 (8), 2010.
[63] Z. Zhang, Q. Wen, and W. Tang, An Efficient Mutation-Based Fuzz Testing Approach for Detecting Flaws of Network Protocol, in: Proc. International Conference on Computer Science & Service System (CSSS), pp.814-817, 2012.
[64] D. Yang, Y. Zhang, and Q. Liu, BlendFuzz: A Model-Based Framework for Fuzz Testing Programs with Grammatical Inputs, in: Proc. IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp.1070-1076, 2012.
[65] P. Ammann, P.E. Black, A Specification-based Coverage Metric to Evaluate Test Sets, International Journal of Reliability, Quality and Safety Engineering 8 (4) (2001) 275-300.
[66] P.E. Black, V. Okun, Y. Yesha, Mutation of Model Checker Specifications for Test Generation and Evaluation, Mutation testing for the new century (Ed. W. E. Wong), Kluwer Academic Publishers, Norwell, Massachusetts, 2001, pp. 14-20.
[67] P.E. Black, V. Okun, Y. Yesha, Mutation Operators for Specifications, in: 15th IEEE International Conference on Automated Software Engineering, ASE 2000, 2000, pp. 81-88.
[68] F. Belli and B. Guldali, Software Testing via Model Checking, Proc. 19th International Symposium on Computer and Information Sciences (ISCIS), Springer-Verlag, pp. 907-916, 2004.
[69] F. Belli and B. Guldali, A Holistic Approach to Test-Driven Model Checking, Proc. 18th Int. Conference on Industrial & Engineering Applications of Artificial Intelligence, LNAI, Vol. 3533, Springer-Verlag, pp. 321-331, 2005.
[70] A.J. Offutt, P. Ammann, and L. Liu, Mutation testing implements grammar-based testing, in Proc. 2nd Workshop Mutation Analysis, Nov. 2006, pp. 12-21.
[71] F. Belli and M. Beyazit, Exploiting Model Morphology for Event-Based Testing, IEEE Transactions on Software Engineering, Vol. 41, Issue 2, 2015, pp. 113-134.
[72] P. Ammann and A.J. Offutt, Introduction to Software Testing, Cambridge Univ. Press, 2008.
[73] F. Belli, M. Beyazit, T. Takagi, and Z. Furukawa, Model-based Mutation Testing Using Pushdown Automata, IEICE Transactions on Information and Systems, E95-D, Vol. 9, 2012, pp. 2211-2218.
[74] F. Belli and M. Beyazit, Mutation of Directed Graphs - Corresponding Regular Expressions and Complexity of Their Generation, in: Proc. 11th Workshop on Descriptional Complexity of Formal Systems (DCFS 2009), Electronic Proceedings in Theoretical Computer Science (EPTCS), Vol. 3, 2009, pp. 69-77.
[75] F. Belli and M. Beyazit, Using Regular Grammars for Event-Based Testing, in: Proc. 18th International Conference on Implementation and Application of Automata, LNCS, Vol. 7982, 2013, Springer-Verlag, pp. 48-59.