Exploiting Model Morphology for Event-Based Testing

Fevzi Belli and Mutlu Beyazıt

Abstract—Model-based testing employs models for testing. Model-based mutation testing (MBMT) additionally involves fault models, called mutants, by applying mutation operators to the original model. A problem encountered with MBMT is the elimination of equivalent mutants and multiple mutants modeling the same faults. Another problem is the need to compare a mutant to the original model for test generation. This paper proposes an event-based approach to MBMT that is not fixed on single events and a single model but rather operates on sequences of events of length $k \geq 1$ and invokes a sequence of models that are derived from the original one by varying its morphology based on k. The approach employs formal grammars, related mutation operators, and algorithms to generate test cases, enabling the following: (1) the exclusion of equivalent mutants and multiple mutants; (2) the generation of a test case in linear time to kill a selected mutant without comparing it to the original model; (3) the analysis of morphologically different models enabling the systematic generation of mutants, thereby extending the set of fault models studied in related literature. Three case studies validate the approach and analyze its characteristics in comparison to random testing and another MBMT approach.

Index Terms—Model-based mutation testing, grammar-based testing, (model) morphology, mutant selection, test generation

1 Introduction

Testing is a user-centric quality assurance technique based on *test cases* that consist of *test inputs* and expected *test behaviors* (commonly characterized by *test outputs*). A *test* invokes the execution or training of the system under consideration (SUC) using a test case. SUC *passes* the test if, upon a test input, the expected behavior is produced; otherwise, the SUC *fails* the test, which then entails the tough *oracle* problem for deriving the expected behavior. A *set of test cases*, also called *test set/suite*, is generated and executed in the target environment of SUC or an environment closely resembling the target environment. Commonly, a *coverage criterion* [68] is used as a stopping condition for testing and providing a measure of the quality of a test set. This paper prefers the term SUC to "system under test (SUT)" because the approach introduced applies both to a model and an implementation, whereas SUT applies to an implementation.

Model-based testing (MBT) is based on creating an abstraction called a *model*, viewing the SUC as a black-box and operating on this model for testing from a behavioral aspect [14]. In *positive testing*, one tests whether the SUC is doing what it is supposed to do; whereas, in *negative testing*, the SUC is tested to determine whether it is not doing what it is not supposed to do [15]. The use of models has various advantages, such as increasing effectiveness and efficiency in terms of fault detection and costs [40]. Formal models additionally help to avoid the oracle problem in the sense that the expected test outputs can automatically be generated [40], [60].

To adopt an MBT approach, a model with a proper expressiveness should be selected based on the SUC and the testing goals. *Expressiveness* (also, *expressive power*) of a model is defined as the breadth of ideas that can be represented and communicated in that model [33]. In general, as expressiveness increases, analyzability decreases [35]. Hence, the use of models with insufficient expressiveness may cause a decrease in the fault detection performance; whereas, the use of models with excessive expressive power may cause an unnecessary increase in the costs.

Some models have the same expressiveness; classical examples are finite state automata (FSA), regular expressions (REs), and regular grammars (RGs) [43], as they relate to the same class of formal languages, that is, type-3 languages [29]. A substantial amount of work in practice relies on the use of such models. Also, pushdown automata (PDA) [43] and event sequence graphs (ESGs) [15] are examples of models having an expressiveness that is, respectively, either stronger or weaker than FSA.

A selected model commonly puts the primary focus on different elements. For example, FSA are state-based; events label the transitions. ESGs and event flow graphs (EFGs) [64], on the other hand, are event-based [19]; they refrain from states and distinguish events from each other by using their contexts. Formal grammars are generally referred to as rule-based models. However, they can be used for both state-based and event-based modeling.

The approach introduced in this paper is event-based. In the context of this paper, the term event is used to mean a discrete action, message, signal, etc. Thus, events are externally perceptible, contrary to states, which are internal to the SUC and thus not necessarily observable [19]. This is the

• *F. Belli is with the Department of Electrical Engineering and Information Technology, University of Paderborn, Paderborn, Germany, and the Department of Computer Engineering, İzmir Institute of Technology, İzmir, Turkey. E-mail: belli@adt.upb.de.*
• *M. Beyazıt is with the Department of Computer Engineering, Yaşar University, İzmir, Turkey. E-mail: mutlu.beyazit@yasar.edu.tr.*

Manuscript received 1 May 2013; revised 29 Jan. 2014; accepted 10 Aug. 2014. Date of publication 1 Oct. 2014; date of current version 4 Feb. 2015. Recommended for acceptance by A. Bertolino.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2360690

reason why this paper chooses formal grammars, the elements of which refer to events that are perceivable to the tester and thus enable him or her to unambiguously decide whether or not the SUC passes the test. Event-based testing operates on sequences of events of increasing length.

Most of the MBT approaches operate on the given model in a fixed way; that is, the model is viewed from only one relevant aspect. However, it is possible to view the same model in different ways to explore morphological differences; for example, an SUC might behave differently to the same input in different contexts. *Morphology* is a Greek word meaning "the study of form or structure." Several disciplines, such as linguistics, chemistry, and astronomy, study the form, structure, or shape of the particular objects of interest. Over the years, the term is also used to refer to *structure*. (This paper does not use the term "structure" to avoid a possible confusion with the term "structural testing" [50].) In MBT, the differences in morphology may cause the associated fault models and the generated test sets to be different.

The *model morphology* this paper exploits is characterized by the length and the contextual relation of the event sequences. By varying the sequence length, the scalability of the approach is also adjusted by algorithmically generating a corresponding sequence of models from the original one. These models describe the same SUC but are morphologically different. This way of model exploitation differs principally from the existing ones; for example, the one used by UML, which creates different kinds of models (diagrams) for different views [57].

Model-based mutation testing (MBMT) [27], [4], [23] is an approach that, in addition to the model given, uses *fault models* for test generation. Thus, MBMT enables both positive and negative testing. *Fault models* are also called *mutants* because they are generated using *mutation operators* that modify the original model. By using mutants, MBT approaches aim to generate test cases which distinguish the mutants from the original model; that is, they *kill* the mutants. When such a test case is executed, the SUC can be tested as to whether or not it contains the fault modeled by the mutant. Evidence suggests that using such model-based mutants is effective at detecting both code-based mutants and real-world faults [6], [10].

MBMT has problems similar to those of (code-based) mutation testing [30], [37], [1] and MBT, because it can be considered as an adaptation of mutation testing using models. For one thing, *some mutants can be equivalent to the original model* or *different mutants can describe the same faults*. This causes a major problem because such mutants lead to the wasting of test resources [2]. Grün, Schuler, and Zeller, among other authors, [36] report that 40 percent of the generated mutants can be equivalent. Furthermore, *each mutant needs to be analyzed against the original model* to detect equivalence or to generate a test case that kills the mutant. However, such an analysis is not always easy (or even possible), because certain models are harder to analyze. In addition, since a fixed model is utilized, *the set of fault models is limited*. This causes certain important faults to be missed.

Formal grammars have already been proposed for MBMT [51], [16], [17]. Building upon these works, this paper introduces a new approach that employs regular grammars for modeling event sequences of length $k \geq 1$ (*k-sequences*), a transformation algorithm to vary model morphology by changing *k*, and related mutation operators to generate corresponding fault models to achieve the following.

- *The generation of only useful mutants.* Existing approaches generate sets of mutants that can include equivalent mutants and multiple mutants that model the same faults. To increase the test efficiency, the attempt is then made to eliminate these mutants. The present approach excludes the generation of such mutants and thus avoids elimination.
- *The generation of a test case in linear time to kill a mutant.* Existing approaches compare each mutant to the original model for test generation. The new approach generates a unique test case to distinguish a selected mutant in linear time without comparing the mutant against the original model.
- *The extension of the set of fault models.* Existing approaches employ a fixed model and, accordingly, generate a set of associated fault models that simply enables the study of the relation between single events. The new approach analyzes the relation between k-sequences and events, enabling the generation of additional fault models, which, in general, represent different or more subtle faults as the sequence length *k* increases. Existing approaches do not consider such fault models.

The paper is organized as follows. Section 2 explains the basic idea behind the approach by way of an example. Section 3 introduces the concepts related to variation of model morphology to extend the set of faults models and to generate test cases. Accordingly, Section 4 discusses strategies for mutant selection from the obtained morphologically different models and test generation from the mutants. Section 5 performs three case studies to analyze the characteristics of the approach in comparison to random testing and *mutate-and-kill-based (MK-based)* MBMT (which is based on generating discriminating test cases). Section 6 discusses the related work. Section 7 concludes the paper and outlines future research.

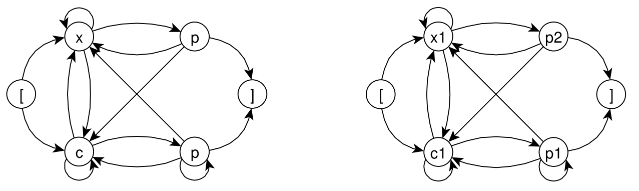
2 Basic Idea Demonstrated by an Example

This section gives an overview of the approach by a simple example. The next three sections explain usage of formal grammars for modeling, mutant generation, and grammar transformation for varying the model morphology. Novelties are exemplified in Section 2.4 and an overview of the approach is given in the last section.

Example 2.1 (Running Example). Consider three events

$$c : \mathbf{copy}, x : \mathbf{cut}, \mathbf{and}\, p : \mathbf{paste}$$

For simplicity, we ignore the operations *select* and *deselect* of system objects or locations. At the beginning, one can perform either c or x. Both c and x can be followed by c, x, or p. If p is performed after c, it can be followed by either c, x, or p. However, if p is performed after x, it can only be followed by either c or x; that is, after cutting and pasting an object, it is not possible to paste it again. One can stop after a p.



S → c1 c(c1) | x1 c(x1)
c(c1) → c1 c(c1) | x1 c(x1) | p1 c(p1)
c(x1) → c1 c(c1) | x1 c(x1) | p2 c(p2)
c(p1) → c1 c(c1) | x1 c(x1) | p1 c(p1) | ε
c(p2) → c1 c(c1) | x1 c(x1) | ε

(a) A model containing ambiguity. (b) A model with contexted events (Ambiguity is removed by indexing). (c) A grammar model which makes use of 1-sequences.

Fig. 1. Event-based models for Example 2.1.

Example 2.1 is a real-world example which is simplified for the sake of readability and saving space; it is used to aid the discussion in the rest of the paper. Also, since the approach is model-based, information on the system internals (such as the source code) is assumed to be not available.

2.1 Event-Based Modeling Using Grammars

Fig. 1a represents an event-based directed graph model to illustrate Example 2.1. Such models are popular in the testing community [19] and have the same expressiveness as FSA. Since the focus of this paper is on events, they are placed at the nodes, and the *follows* relation between the events is visualized by arcs. Pseudo-events [and] are used to mark, respectively, the start and finish events [15].

The model in Fig. 1a has a severe drawback. By "event *p*," one cannot differentiate to which *p* event is referred. The present approach suggests distinguishing such events from each other by indexing that considers the *context*s in which they reside, leading to *contexted events*, such as $\{c1, x1, p1, p2\}$ (Fig. 1b). Their counterparts, *basis events*, such as *{c, x, p}*, represent the events as they are visible to the user. Note that contexted events are not necessarily caused by cycles or loops in the model.

Fault models associated with the models like the one in Fig. 1b are primarily based on modifying the *follows* relation between single events. This modification needs to be generalized by analyzing occurrences of single events with respect to event sequences of length $k \geq 1$ (*k-sequences*) for systematic extension of event-based fault modeling.

Grammars are suitable for representing event-based abstractions based on k-sequences. They allow multiple occurrences of events in *productions*, which enable to represent the *follows* relation between k-sequences and events. This practice is common in compiler construction and testing [43]; related techniques are exploited here.

In light of the discussion above, the grammar model is composed of a set of *(contexted) events*, a set of *basis events*, a set of *k-sequences (terminals)*, a set of *contexts (nonterminals)* including a *start context* and a set of *productions*. A context relation determines the right unique context of a k-sequence in productions.

Example 2.2 (Grammar Model). Fig. 1b shows the indexed version of Fig. 1a where the contextual ambiguity of *p* is eliminated. For a unified representation, unambiguous events are also indexed. Based on Figs. 1b, 1c represents the grammar that precisely models Example 2.1. The productions have the following semantics.

- $c(a) \rightarrow b$ *c(b)* means that *b* follows *a* and *a b* is a 2-sequence.
- $S \rightarrow a$ *c(a)* means that *a* is a start event.
- $c(a) \rightarrow \varepsilon$ means that *a* is a finish event.

Also, *c(a)* denotes the (right) context of event *a*.

The productions of Example 2.2 form a regular grammar. The terminals therein are events that can be viewed as 1-sequences, and the nonterminals are contexts. Therefore, this model is called "1-Reg."

2.2 Generating Mutants

The new approach refines the elementary mutation operators *insertion* and *omission* [23] to modify sequences of events by also considering the *start* and *finish* events. The iterative and combinatorial deployment of these operations enables further mutation operators such as *duplication*, *deletion*, or *replacement* [51], [9].

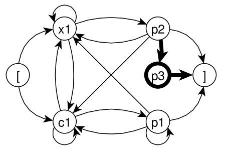
Example 2.3 (Mutants). Fig. 2 contains some mutants of Example 2.2. The mutant in Fig. 2a is generated using an event-based mutation [23] by inserting event/terminal *p3*. Furthermore, the mutant in Fig. 2b is generated using a grammar-based mutation [51], [9] by replacing terminal *p1* by *x1*.

These mutants are different. Fig. 2a is a 1-Reg; it models a single fault: "*p* is extra after *x p*." In contrast, Fig. 2b is not a 1-Reg but an RG; it models multiple faults: "*p* is missing after *c*," and "*p* is extra after *c x p*."

2.3 Grammar Transformation to Vary Morphology

The introduced event-based grammar model enables the generation of morphologically different models by a transformation to vary *k*.

Example 2.4 (Transformed Model). The model in Fig. 1c and its transformation shown in Fig. 3 describe the same system, but productions in Fig. 3 utilize 2-sequences; therefore, it is a "2-Reg." A 2-Reg production of the form $c(a\,e) \rightarrow e\;b\;c(e\;b)$ means that *b* follows *a e* and *a e b* is a 3-sequence.



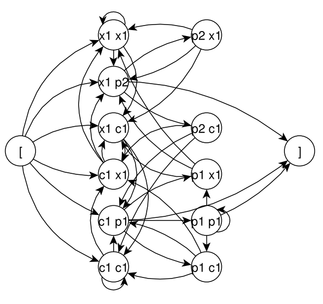
S → c1 c(c1) | x1 c(x1)
c(c1) → c1 c(c1) | x1 c(x1) | **x1** c(p1)
c(x1) → c1 c(c1) | x1 c(x1) | p2 c(p2)
c(p1) → c1 c(c1) | x1 c(x1) | p1 c(p1) | ε
c(p2) → c1 c(c1) | x1 c(x1) | ε

(a) An insert event mutant. (b) A terminal replacement mutant.

Fig. 2. Some mutants of the model in Fig. 1c (Mutations are boldfaced or underlined).

S → c1 c1 c(c1 c1) | c1 x1 c(c1 x1) | c1 p1 c(c1 p1) |
    x1 c1 c(x1 c1) | x1 x1 c(x1 x1) | x1 p2 c(x1 p2)
c(c1 c1) → c1 c1 c(c1 c1) | c1 x1 c(c1 x1) | c1 p1 c(c1 p1)
c(c1 x1) → x1 c1 c(x1 c1) | x1 x1 c(x1 x1) | x1 p2 c(x1 p2)
c(c1 p1) → p1 c1 c(p1 c1) | p1 x1 c(p1 x1) | p1 p1 c(p1 p1) | ε
c(x1 c1) → c1 c1 c(c1 c1) | c1 x1 c(c1 x1) | c1 p1 c(c1 p1)
c(x1 x1) → x1 c1 c(x1 c1) | x1 x1 c(x1 x1) | x1 p2 c(x1 p2)
c(x1 p2) → p2 c1 c(p2 c1) | p2 x1 c(p2 x1) | ε
c(p1 c1) → c1 c1 c(c1 c1) | c1 x1 c(c1 x1) | c1 p1 c(c1 p1)
c(p1 x1) → x1 c1 c(x1 c1) | x1 x1 c(x1 x1) | x1 p2 c(x1 p2)
c(p1 p1) → p1 c1 c(p1 c1) | p1 x1 c(p1 x1) | p1 p1 c(p1 p1) | ε
c(p2 c1) → c1 c1 c(c1 c1) | c1 x1 c(c1 x1) | c1 p1 c(c1 p1)
c(p2 x1) → x1 c1 c(x1 c1) | x1 x1 c(x1 x1) | x1 p2 c(x1 p2)

(a) Productions.



(b) Directed graph visualization.

Fig. 3. A grammar model for Example 2.1 which makes use of 2-sequences (Transformed from Fig. 1c).

2.4 Novelties

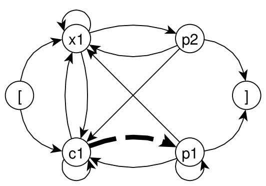
The set of fault models is extended. To see how morphologically different models, generated using grammar transformation, extend the set of possible fault models, consider a mutant of Fig. 3 generated by omitting sequence *(p1 c1, c1 p1)* as shown in Fig. 4b. This mutant models the fault that

$$p1 \textbf{ is missing after } p1\ c1;$$

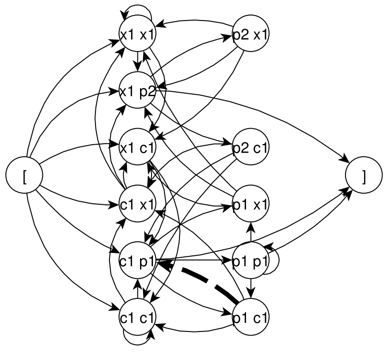
that is, paste fails after performing a paste and a copy. It is not possible to create such a mutant from the model in Fig. 1c by a simple omission. For example, one can omit sequence *(c1, p1)* (see Fig. 4a). However, in this mutant, paste fails immediately after performing a copy. Hence, the mutant in Fig. 4b models a different and more subtle fault than the mutant in Fig. 4a. Thus, the set of fault models can be extended by generating mutants modeling different or more subtle faults.

To the knowledge of the authors, no other existing approach directly considers such a fault.

Only useful mutants are generated. Most of the MBMT approaches, such as [8], [4], compare each mutant against the original model to check if they are equivalent. In contrast, the proposed approach excludes equivalent mutants and multiple mutants modeling the same faults [17].



(a) A mutant of the model in Fig. 1c.



(b) A mutant of the model in Fig. 3.

Fig. 4. Two mutants for Example 2.1 (Mutations are shown in boldface dashed lines).

Each selected mutant has the following properties. (1) It does not violate the type-3ness of the given grammar; that is, the mutated grammar is of the same type as the original one (Also, see the discussion in Section 6.3). (2) It models a small number of faults. (3) The faults are located at the mutation point; that is, the faults are directly related to the mutation parameter.

The mutant in Fig. 2a is selected because it is a 1-Reg (the type is preserved), it models a single fault where *p* is extra after *p2* (or after *x p*), and the fault is located at the mutation point because the inserted event is itself faulty.

The mutant in Fig. 2b is excluded because it models multiple faults which can be modeled separately. *p* is missing after *c*, and *p* is extra after *c x p*.

A test case is generated in linear time to kill a mutant. Since the location of the faults modeled by each selected mutant can be determined from the actual mutation parameter, a unique test case to kill the mutant can be generated in linear time, without comparing it against the original model. For example, breadth-first search can be used to generate *x1 p2 p3* to kill the mutant in Fig. 2a.

2.5 An Overview of the Proposed Approach

The proposed approach follows the steps below.

1) Create the initial model of the SUC, a 1-Reg (Section 3.1).
2) Vary the morphology of the 1-Reg by transforming it into a k-Reg for some integer *k* (Section 3.2).
3) Generate a set of positive test cases using the k-Reg for detection of missing event faults (Section 3.3).
4) Leave out the equivalent mutants and multiple mutants modeling the same faults using the mutant selection strategies and select a subset of all possible mutants of the k-Reg (Section 4.1 and Section 4.2).
5) Use the selected mutants to generate a set of negative test cases for detection of extra event faults (Section 4.3).

3 Varying Morphology to Extend the Set of Fault Models

This section discusses the notions and concepts, starting with basic notions. One of the key concepts, grammar transformation to vary the morphology, follows before the generation of positive test cases that concludes the section. Note that generation of negative test cases is discussed in Section 4, combined with mutant generation.

3.1 Basic Notions

The grammar model has informally been introduced in Section 2.1. The formal definition follows.

Definition 3.1 (k-Sequence Right Regular Grammar (k-Reg)). *A k-sequence right RG (integer $k \geq 1$) is a quintuple $G = (E, B, K, C, P)$ where:*

- *E is a finite set of events (or contexted events).*
- *B is a finite set of basis events, which is the set of all visible events under consideration. For $e \in E, d(e) \in B$ is the corresponding basis event (the noncontexted version of e), and d(.) is the decontexting function.*
- *$K \subseteq E^k$ is a finite set of k-sequences (or terminals). For $r \in K, r = r_1 \dots r_k$ and $d(r) = d(r_1) \dots d(r_k) \in B^k$ is the corresponding basis k-sequence.*
- *C is a finite set of contexts (or nonterminals) where*
- *$S \in C$ is the start context (or start symbol).*
- *P is a finite set of productions of the form*

$$Q \to \varepsilon \text{ or } Q \to r\, c(r)$$

where $Q \in C$ is a context, $r \in K$ is a k-sequence, $c(r) \in C \setminus \{S\}$ is the unique context of r, and ε is the empty string. If $k \geq 2$, then for each $c(q) \to r\, c(r) \in P$ where $q = q_1 \dots q_k \in K$ and $r = r_1 \dots r_k \in K$,

$$q_2 \dots q_k = r_1 \dots r_{k-1}.$$

Note that k-sequences are defined as terminals and have different, therefore, unique, contexts. The semantics of the productions is as follows. For each $c(q) \to r\, c(r) \in P$, *r_k follows q in the system modeled by grammar G*; that is, $q\, r_k$ is a (k+1)-sequence in the system. Also, r is a *start k-sequence* for each $S \to r\, c(r) \in P$, and q is a *finish k-sequence* for each $c(q) \to \varepsilon \in P$. These productions allow only right linearity, ensuring type-3 preservation.

Productions of a k-Reg can be visualized via directed graphs by labeling nodes using the k-sequences and [and]. Arcs of the form "([, r)", "(r,])", and "(q, r)"correspond to the productions of the form "$S \to r\, c(r)$", "$c(r) \to \varepsilon$", and "$c(q) \to r\, c(r)$", respectively.

Productions of a k-Reg are used to derive strings. A *derivation*, denoted by $\Rightarrow_G{}^*$, is a sequence of *derivation steps*, each of which is of the form $xQy \Rightarrow_G xRy$ where $x,y \in (C \cup K)^*$ and $Q \to R \in P(\Rightarrow^*$ and $\Rightarrow$ are used when there is no confusion). The number of derivation steps in a derivation is called *the length of the derivation*. Also, the *language* defined by grammar G is the set of strings $L(G) = \{w | S \Rightarrow^* w (w \in K^*)\}$.

The example that was informally given in Section 2.1 can be formalized as follows.

Example 3.1 (A 1-Reg). Below, 5-tuple is a 1-Reg, which describes Example 2.1 using 1-sequences.

- $E = \{c1, x1, p1, p2\}$
- $B = \{c, x, p\}$ where $c = d(c1)$, $x = d(x1)$ and $p = d(p1) = d(p2)$.
- $K = E$, since $k = 1$.
- $C = \{S, c(c1), c(x1), c(p1), c(p2)\}$.
- S designates the initial point.
- P contains 15 productions (see Fig. 1c or Fig. 1b).

Function $d(.)$ (in Definition 3.1) can be extended to associate (contexted) sequences with basis sequences. For an event sequence $s = s_1\, s_2 \dots s_n$, *the corresponding basis event sequence of s* is $d(s) = d(s_1)\, d(s_2) \dots d(s_n)$, where $d(\varepsilon) = \varepsilon$. Furthermore, *the corresponding set of basis event sequences of a set of event sequences X* is $d(X) = \{d(s) \mid s \in X\}$.

Example 3.2 (Decontexted Event Sequences). Consider the 1-Reg in Fig. 1c. For event sequence $s = c1\; x1\; p2\; c1\; p1\; p1$, $d(s) = c\; x\; p\; c\; p\; p$. Also, for set of event sequences $X = \{c1, c1\; p1, c1\; x1\; p2\}$, $d(X) = \{c, c\; p, c\; x\; p\}$.

Event sequences that can and cannot be derived using k-Reg productions are distinguished for testing. For a k-Reg $G = (E, B, K, C, P)$, an event sequence s is said to be *in grammar G*, if it can be derived using some productions in P. A nonempty event sequence s in G is a *start* [or *finish*] *sequence*, if there is a derivation of the form $S \Rightarrow^* s\, Q$ $(Q \in C)$[or $Q \Rightarrow^* s$ $(Q \in C)$]. An event sequence which is not in G is also called a *faulty event sequence*.

Example 3.3 (Event Sequences in a 1-Reg). For the 1-Reg in Fig. 1c:

- 2-sequences in $\{c1\; x1, x1\; p2, p1\; p1\}$ are in the 1-Reg, whereas 2-sequences in $\{p2\; p1, p2\; p2\}$ are not.
- $\{c1, x1, c1\; c1, x1\; x1\; p2, c1\; p1\; x1\}$ is a set of start sequences, and $\{p1, p2, p1\; p1, x1\; p2\}$ is a set of finish sequences.

By Definition 3.1, one can obtain a (k×m)-sequence s using a derivation of length $m \geq 1$, so that $s \in K^*$ and s is in G. These sequences are important for testing and called *m-derived sequences*. Each sequence in G appears in such a sequence.

Example 3.4 (A 3-derived Sequence in a 2-Reg). $p1\; x1\; x1\; p2\; p2\; c1$ is a 3-derived sequence for the 2-Reg in Fig. 3; it is derived using $c(c1\; p1) \to p1\; x1\; c(p1\; x1)$, $c(p1\; x1) \to x1\; p2\; c(x1\; p2)$ and $c(x1\; p2) \to p2\; c1\; c(p2\; c1)$.

In event-based testing, k-Regs and their mutants are used to generate positive and negative test cases. The aim is to reveal *missing event faults* where an event cannot occur after or before a (possibly empty) sequence of events and *extra event faults* where an event can occur after or before a (possibly empty) sequence of events.

Definition 3.2 (Positive and Negative Test Cases). *Given a k-Reg $G = (E, B, K, C, P)$.*

- *An event sequence is a positive test case, if it is a start sequence in G, or it is ε. $T_P(G)$ denotes the set of all positive test cases. A complete event sequence (CES) is a positive test case which is both a start and a finish sequence in G, or it is ε if $\varepsilon \in L(G)$. $T_{CES}(G) = L(G) \subseteq T_P(G)$ denotes the set of all CESs.*
- *An event sequence is a negative test case, if the first event in it is a nonstart event or it contains at least one 2-sequence which is not in G. $T_N(G)$ denotes the set of all negative test cases. A faulty complete event sequence (FCES) is a negative test case which either is composed of only a nonstart event, or contains only one 2-sequence which is not in G and it ends with this 2-sequence. $T_{FCES}(G) \subseteq T_N(G)$ denotes the set of all FCESs.*
- *A set of test cases is also called a test set.*

Example 3.5 (Test Cases of a 1-Reg). For the 1-Reg in Fig. 1c

- $\{c1, x1\ x1, c1\ p1\ p1\ x1\}$ is a set of positive test cases, and $\{x1\ p2, x1\ x1\ p2, c1\ p1\ p1\ p1\}$ is a set of CESs.
- $\{p1, x1\ p2\ p1\ c1, c1\ x1\ p2\ p2\}$ is a set of negative test cases, and $\{x1\ p2\ p2, c1\ x1\ p2\ p2\}$ is a set of FCESs.

Each event in a given k-Reg is contexted. However, different occurrences of system behavior are based on basis events since they correspond to system events visible to the user. Thus, the equivalence of two k-Regs is defined as follows.

Definition 3.3 (Equivalence). *Two k-Regs G and H are equivalent, if $d(T_{CES}(G)) = d(T_{CES}(H))$.*

In practice, it is important that all k-sequences in a k-Reg are utilized, in other words, that they are useful.

Definition 3.4 (Usefulness). *Given a k-Reg $G = (E, B, K, C, P)$. A string $z \in (C \cup E)^*$ is useful in grammar G, if $S \Rightarrow^* xzy \Rightarrow^* w$ for some $x,y \in (C \cup E)^*$ and $w \in E^*$. G is useful, if all k-sequences in K are useful in G.*

Example 3.6 (A Useful and a Nonuseful 1-Reg). k-Regs in Figs. 1c and 3 are all useful. To obtain a non-useful 1-Reg from Fig. 1c, one can remove $c(p1) \to \varepsilon$ and $c(p2) \to \varepsilon$. The resulting grammar does not have any finish events. Therefore, $T_{CES}(G)$ is empty, but the follows relation is still described correctly.

Deterministic system models help to exclude redundant event sequences from the model.

Definition 3.5 (Determinism). *A k-Reg $G = (E, B, K, C, P)$ is deterministic, if, for each $Q \in C$, there are no two productions $Q \to q\, c(q) \in P$ and $Q \to r\, c(r) \in P$ such that $r \neq q$ and $d(r) = d(q)$.*

Example 3.7 (Test Cases of a Deterministic 1-Reg). 1-Reg obtained by including *c(p1)→ p2 c(p2)* in Fig. 1c is nondeterministic. Positive test cases $s = c1\ p2\ c1$ and $t = c1\ p1\ c1$ are redundant because *d(s) = d(t)*.

Unless noted otherwise, all grammars under consideration are useful and deterministic k-Regs.

3.2 Grammar Transformation to Vary Morphology

Based on Definition 3.1, a (k+1)-Reg model is morphologically different from a k-Reg model, and it can be used to model different or more subtle faults. To do this, a transformation to vary k and generate models with morphological differences is constructed.

To give the definition of k-Reg transformation, the following observations are generalized:

- For each $c(q_1\ q_2) \to r_1\ r_2\ c(r_1\ r_2)$ in Fig. 3, $q_2 = r_1$. Thus, each such production can be obtained by using $c(q_1) \to q_2\ c(q_2)$ and $c(q_2) \to r_2\ c(r_2)$ in Fig. 1c.
- Each $S \to r_1\ r_2\ c(r_1\ r_2)$ in Fig. 3 can be obtained by using $S \to r_1\ c(r_1)$ and $c(r_1) \to r_2\ c(r_2)$ in Fig. 1c.
- Each $c(r_1\ r_2) \to \varepsilon$ in Fig. 3 can be obtained by using $c(r_1) \to r_2\ c(r_2)$ and $c(r_2) \to \varepsilon$ in Fig. 1c.

Definition 3.6 (k-Reg Transformation). *Given a 1-Reg $G_1 = (E, B, K_1, C_1, P_1)$.*

- *The corresponding 1-Reg of G_1 is itself: G_1.*
- *Let $G_k = (E, B, K_k, C_k, P_k)$ be the corresponding k-Reg of G_1. The corresponding (k+1)-Reg of G_1* (or G_k) *is*
- *$G_{k+1} = (E, B, K_{k+1}, C_{k+1}, P_{k+1})$ where:*
  - *$K_{k+1} = \{q_1 \ldots q_k\ r_k | c(q) \to rc(r) \in P_k\}$ is the set of (k+1)-sequences in G_1.*
  - *$C_{k+1} = \{c(r) | r \in K_{k+1}\}$ is the set of contexts.*
  - *$P_{k+1} = \{S \to r\, e\, c(r\ e) | S \to r\ c(r) \in P_k$ and $c(r_k) \to e\ c(e) \in P_1\} \cup \{c(q\ r_k) \to \varepsilon | c(q) \to r\ c(r) \in P_k$ and $c(r_k) \to \varepsilon \in P_1\} \cup \{c(q\ r_k) \to r\, e\, c(r\ e) | c(q) \to r\ c(r) \in P_k$ and $c(r_k) \to e\ c(e) \in P_1\}$ is the set of productions.*

Based on Definition 3.6, Algorithm 1 performs k-Reg transformation. It runs in $O(k|P_1||P_k|) = O(k|P_1|^{k+1})$ worst case time because (1) $|P_k| = O(|P_1|^k)$; (2) All set union operations can be performed in $O(1)$ time because a different element is added during each union; (3) A (k+1)-sequence in G_1 can be constructed in $O(k)$ steps by merging a k-sequence in G_1 extracted from G_k with a 1-sequence in G_1 extracted from G_1. This time complexity is expected because the number of k-sequences increases exponentially in k. In practice, however, $|P_k|$ is generally much smaller than $|P_1|^k$, and k is almost always bounded. Hence, transformation can be performed quite fast.

Algorithm 1. k-Reg Transformation

```
Input: $G_k = (E, B, K_k, C_k, P_k)$—the corresponding k-Reg of $G_1$
    $G_1 = (E, B, K_1, C_1, P_1)$—the 1-Reg
Output: $G_{k+1} = (E, B, K_{k+1}, C_{k+1}, P_{k+1})$—the corresponding
    (k+1)-Reg
  $K_{k+1} = \emptyset, C_{k+1} = \{S\}, P_{k+1} = \emptyset$
  for each $Q \to r\ c(r) \in P_k$ where $r = r_1 \ldots r_k$ do
    if $Q = c(q)$ where $q = q_1 \ldots q_k$ then
      $K_{k+1} = K_{k+1} \cup \{q\ r_k\}, C_{k+1} = C_{k+1} \cup \{c(q\ r_k)\}$
    end if
    for each $c(r_k) \to R \in P_1$ do
      if $R = e\ c(e)$ then
        if $Q = S$ then
          $P_{k+1} = P_{k+1} \cup \{S \to r\ e\ c(r\ e)\}$
        else if $Q = c(q)$ then
          $P_{k+1} = P_{k+1} \cup \{c(q\ r_k) \to r\ e\ c(r\ e)\}$
        end if
      else if $R = \varepsilon$ then
          $P_{k+1} = P_{k+1} \cup \{c(q\ r_k) \to \varepsilon\}$
      end if
    end for
  end for
```

Example 3.8 (A Corresponding 2-Reg). Grammar in Fig. 3 is the corresponding 2-Reg of the 1-Reg in Fig. 1c.

As demonstrated in Section 2.4 (Also, Fig. 4), generated k-Reg models can be used to extend the set of fault models due to their morphological differences.

A sequence in the corresponding k-Reg of a given 1-Reg need not be a sequence in this 1-Reg. However, this sequence can be used to obtain a sequence in the 1-Reg by using the following definition and theorem.

Definition 3.7 (Sequence Transformation). *Given a $(k \times m)$-sequence $s = u^1 \ldots u^m$ where $k \geq 1, m \geq 1$ and $u^i = u^i_1 \ldots u^i_k$ for $i = 1, \ldots, m$. Inverse sequence transformation of s based on integer k is defined as a $(k+m-1)$-sequence*

$$T_S^{-1}(s,k) = u^1 u_k^2 u_k^3 \ldots u_k^m$$

where $u^1 = s_1 \ldots s_k$ and each $u_k^i = s_i$ for $i = 2, \ldots, m$.

Theorem 3.1 (From m-derived Sequences in a Corresponding k-Reg to (k+m−1)-derived Sequences in 1-Reg). *Given a 1-Reg G_1, and its corresponding k-Reg G_k where $k \geq 2$ and $K_k \neq \emptyset$. If s is an m-derived sequence in G_k, $t = T_S^{-1}(s,k)$ is a $(k+m-1)$-derived sequence in G_1.*
(Proof is included in Appendix.)

Example 3.9 (Sequence Transformation). $T_S{}^{-1}(s,2) = c1\ c1\ x1\ x1\ p2$ is a 5-derived sequence in the 1-Reg in Fig. 1c for four-derived sequence $s = c1\ c1\ c1\ x1\ x1\ x1\ x1\ p2$ in the 2-Reg in Fig. 3.

3.3 Positive Test Case Generation

A corresponding k-Reg can be used to generate a test set that covers all (k+1)-sequences. In this way, one can reveal missing event faults where an event does not follow a certain k-sequence.

By Definition 3.1 and Definition 3.6, a corresponding k-Reg contains all (k+1)-sequences in its productions. Hence, one can cover the productions to generate a set of sequences and use Theorem 3.1 to obtain a test set covering all (k+1)-sequences (see Algorithm 2).

Algorithm 2. Achieving (k+1)-sequence Coverage

```
Input: $G_k = (E, B, K, C, P)$—the k-Reg ($k \geq 1$)
Output: $X$ – a set of sequences covering all (k+1)-sequences
  $X = \emptyset$
  $Y$ = generate a set covering all productions of $G_k$
  for each $s \in Y$ do
    $X = X \cup T_S{}^{-1}(s,k)$//see Theorem 3.1
  end for
```

The time complexity of Algorithm 2 is given by $O(C_P(|E|,|P|) + C_T(|E|,|P|)) = O(C_P(|E|,|P|))$ where (1) $C_P(|E|,|P|)$ is the time complexity of generating a set of sequences achieving production coverage for G_k; and (2) $C_T(|E|,|P|)$ is the time complexity of inverse transforming these sequences to obtain test cases. Although there is no detailed time complexity analysis for fast grammar-based test generation algorithms [56], [46], [66], the performance is generally polynomial in $|P|$, that is, $O(|P|^c)$ for some $c \geq 1$. For example, even if each production is covered a minimum number of times, the complexity becomes $O(|K|^3) = O(|P|^3)$ [22]. Using such algorithms helps to generate reduced test sets.

Example 3.10 (A Test Set Generated Using Algorithm 2). To achieve 2-sequence coverage for the 1-Reg in Fig. 1c, this 1-Reg can be used as input to Algorithm 2. The following is an example test set:

$$\{c1\,c1\,x1\,c1\,p1\,c1\,p1\,x1\,x1\,p2\,c1\,p1\,p1,\ \ x1\,p2\,x1\,p2,\ c1\,p1\}.$$

4 Mutant Selection to Increase Efficiency

The approach selects mutants that are of the same type as the original model. They model a small number of faults, which are located at the mutation points so that one modeled fault does not interfere with another. This section shows that there is no need to compare each mutant to the original model for equivalence or test generation, and the generation of equivalent mutants and multiple mutants modeling the same faults can be avoided. Also, a test case to kill the mutant can be generated in linear time.

Although various different mutation operators ([23], [51], [9]) can be defined for k-Regs to model missing and extra event faults, only some of these operators are needed due to the following assumptions, considering the fact that mutants are used in test generation.

A1. Events in a test case are executed in the given order until a failure is observed.
A2. A test case can end with any event, which needs not be a finish event.

Thus, for a given k-Reg, the following can be stated.

P1. Missing and extra event faults are limited by considering the k-sequences that precede the missing or extra events while ignoring the succeeding k-sequences. Thus, by exercising all (k+1)-sequences in the k-Reg, one can test whether an event is missing after some k-sequence, and, by exercising all relevant faulty k-sequences, one can test whether an event is extra after some k-sequence. (By A1)
P2. Mark nonstart, mark nonfinish, omit sequence, and omit terminal mutants are discarded because they do not contain any (k+1)-sequences that are not contained in the original model. (Due to P1)
P3. Mark finish and mark nonfinish mutants do not really correspond to fault models because every event can be considered as a finish or nonfinish event during the testing process. (By A2)
P4. Faults modeled using insert sequence mutants can be modeled using insert terminal mutants. (By definition [23])
P5. Nonterminal and terminal duplication, deletion and replacement mutants are discarded because they contain multiple missing event or extra event faults. Also, nonterminal replacement is not type-preserving (see Section 6.3). (By definition [51], [9])
P6. All negative test cases are FCES. (By A1)

Consequently, one can use the original k-Reg to cover (k+1)-sequences for missing event faults (as outlined in Section 3.3) and mark start and insert terminal mutants to cover faulty 1-sequences and faulty (k+1)-sequences for extra event faults. Thus, only mark start and insert terminal operators are studied by proposing mutant selection strategies and developing test generation methods.

G = (E, B, K, C, P) is considered as the original k-Reg model in the discussion, unless noted otherwise.

4.1 Mark Start Mutant Selection

Mark start mutation operator is used to mark k-sequences as start k-sequences. Therefore, mark start mutants are used to model extra start event faults.

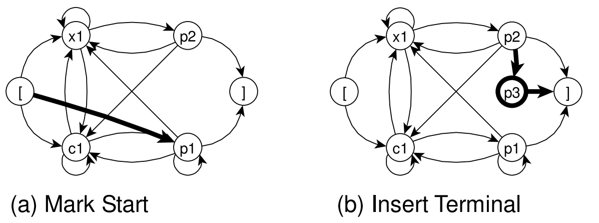


Fig. 5. Mark start and insert terminal mutants of the 1-Reg in Fig. 1c (Mutations are drawn using boldface).

Definition 4.1 (Mark Start). *Given a k-sequence $e \in K$ such that $S \rightarrow e\ c(e) \notin P$, mark start (Ms) operator is defined as $Ms(G,e) = (E,B,K,C,P')$ where $P' = P \cup \{S \rightarrow e\ c(e)\}$.*

In the following, $G' = Ms(G,e) = (E,B,K,C,P')$, unless noted otherwise.

Example 4.1 (A Mark Start Mutant). Let G be the 1-Reg in Fig. 1c. Fig. 5a shows *Ms(G, p1)*.

The set of all CESs is extended due to the mutation.

Lemma 4.1 (Set of CESs of a Mark Start Mutant). *The set of all CESs of G' is given by*

$$T_{CES}(G') = T_{CES}(G) \cup \{e\ x \mid c(e) \Rightarrow_G^* x (x \in E^*)\}.$$

(Proof is included in Appendix.)

Example 4.2 (Set of CESs of a Mark Start Mutant). Marking *p1* as a start event in 1-Reg Fig. 1c (see Fig. 5a) extends the set of CESs of the 1-Reg. Event sequences that start with p1, such as *p1*, *p1 p1*, *p1 c1 p1* and *p1 x1 x1 p2*, are included in the new set.

Using Lemma 4.1, Lemma 4.2 discusses the equivalence of a mark start mutant to the original k-Reg.

Lemma 4.2 (Equivalence of a Mark Start Mutant). *G' is not equivalent to G if and only if $d(X) \backslash d(Y) \neq \emptyset$ where*

- *$X = \{e\ x \mid c(e) \Rightarrow_G {}^* x\ (x \in E^*)\}$ and*
- *$Y = \{e'\ y \mid S \Rightarrow_G {}^* e' y\ (e' \in K, y \in E^*)$ where $e' \neq e$ and $d(e') = d(e)\} \subseteq T_{CES}(G)$.*

(Proof is included in Appendix.)

Example 4.3 (Equivalence of a Mark Start Mutant). Let G be the 1-Reg in Fig. 1c and $G' = Ms(G, p1)$ be the 1-Reg in Fig. 5a. G' extends $T_{CES}(G)$ by X that contains new sequences starting with *p1*. Since there is no CES starting with a p event in G, $Y = \emptyset$. Thus, $d(X) \backslash d(Y) = d(X) \neq \emptyset$, which means that there are additional decontexted event sequences in $d(T_{CES}(G'))$. Thus, $d(T_{CES}(G')) \neq d(T_{CES}(G))$ and G' is not equivalent to G.

Sufficient conditions for usefulness, determinism, and nonequivalence of a mark start mutant are outlined in the following.

Theorem 4.1 (Usefulness and Determinism of a Mark Start Mutant). *G' is useful, if G is useful, and G' is deterministic, if G is deterministic and there is no $S \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$.*
(Proof is included in Appendix.)

Example 4.4 (Usefulness and Determinism of a Mark Start Mutant). Let G be the 1-Reg in Fig. 1c. Since G is useful, $G' = Ms(G, p1)$ in Fig. 5a is also useful. Furthermore, since G is deterministic and no p event is a start event in G, G' is also deterministic.

Theorem 4.2 (Nonequivalence of a Mark Start Mutant). *G' is not equivalent to G, if G is useful and there is no $S \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$.*
(Proof is included in Appendix.)

Example 4.5 (Nonequivalence of a Mark Start Mutant). G, the 1-Reg in Fig. 1c is useful and it has no p event as a start event. Therefore, its mutant $G' = Ms(G, p1)$, the 1-Reg in Fig. 5a, is not equivalent to G.

The mutant selection strategy is now devised for mark start mutants.

Mark Start Mutant Selection. *Given a k-Reg $G = (E, B, K, C, P)$. For each Ms(G, e), k-sequence e is selected as a mutation parameter if the following hold:*

1. *There is no start k-sequence x such that $d(x_1) = d(e_1)$.*
2. *There is no previously selected mutation parameter y such that $d(y_1) = d(e_1)$.*

Let G be a useful and deterministic k-Reg. By Theorem 4.1 and Theorem 4.2, mutants generated from G using the above strategy are useful, deterministic, and nonequivalent to G. Furthermore, each of these mutants models a different fault located at the mutation point; that is, e_1 (also $d(e_1)$) is an extra start event for each $Ms(G,e)$.

The left-out mark start mutants are useful. However, they are either nondeterministic or model previously modeled faults. Some nondeterministic mutants do not model any extra event faults. If they do, these faults are not extra start event faults; therefore, they can be modeled using insert terminal mutants.

Algorithm 3 selects mark start mutants using the above strategy. Its runtime complexity is given by $O(|B||P|)$: (1) The number of mutants generated is bounded by $|B|$ because each mutant represents a different extra start event fault; and (2) Each mutant *Ms(G, e)* can be generated in $O(|P|+|B|) = O(|P|)$ time by checking if there are no start k-sequence x so that $d(x_1) = d(e_1)$ and previously selected mutation parameter y so that $d(y_1) = d(e_1)$, and copying G to modify it.

Algorithm 3. Mark Start Mutant Selection

```
Input: $G = (E, B, K, C, P)$—the k-Reg
Output: $M$—the set of selected mark start mutants
  $M = \emptyset, N = \emptyset$
  for each $b \in B$ do
    if there is no $S \rightarrow x\ c(x) \in P$ such that $d(x_1) = b$ and
      there is no $y \in N$ such that $d(y_1) = b$ then
       Select a k-sequence $e \in K$ such that $d(e_1) = b$
      $G' = G, M = M \cup \{Ms(G', e)\}, N = N \cup \{e\}$
    end if
  end for
```

Also, from each selected mutant, a unique test case that kills it can be generated in $O(1)$ time by simply taking e_1.

Example 4.6 (Mark Start Mutant Selection). Let G be the 1-Reg in Fig. 1c. The only selected mark start mutant is $Ms(G, p1)$. $Ms(G, c1)$ and $Ms(G, x1)$ are excluded because *c1* and *x1* are already start events. Furthermore, $Ms(G, p2)$ is excluded because it models the same fault as $Ms(G, p1)$.

4.2 Insert Terminal Mutant Selection

Insert terminal mutation operators are used to add new terminals (k-sequences) by (possibly) connecting them to the existing k-sequences. Therefore, insert terminal mutants are used to model extra event faults where an event follows some k-sequence.

Definition 4.2 (Insert Terminal). *Given a k-sequence $e \notin K$ such that $d(e) \in B^k$, $U = \{(a,e) \mid a \in \{a_1,\ldots,a_m\} \subseteq K\}$ and $V = \{(e,b) \mid b \in \{b_1,\ldots,b_n\} \subseteq K \cup \{e\}\}$, insert terminal (It) operator is defined as It $(G,e,U,V) = (E,B,K',C',P')$ where $K' = K \cup \{e\}$, $C' = C \cup \{c(e)\}$, and $P' = P \cup \quad \{c(e) \to \quad a\ c(a) \mid (a,e) \in U\} \cup \{c(b) \to e\ c(e) \mid (e,b) \in V\}$.*

To generate mutants that contain a small number of changes, $|U| = 1$, that is, $U = \{(a,e)\}$. Furthermore, since all negative test cases are FCESs, $V = \emptyset$ and $c(e) \to \varepsilon$ is inserted for the usefulness of k-sequence *e*. Therefore, in the following, $G' = It(G,e,\{(a,e)\},\emptyset) = (E,B,K',C',P')$, unless noted otherwise.

Example 4.7 (An Insert Terminal Mutant). Let *G* be the 1-Reg in Fig. 1c. Fig. 5b shows $It(G,p3,\{(p2,p3)\},\emptyset)$ where *p3* is a new contexted paste event. Note that, since $V=\emptyset, c(d1) \to \varepsilon$ is additionally inserted to preserve usefulness of *p*3.

Lemma 4.3 (Set of CESs of an Insert Terminal Mutant). *The set of all CESs of G′ is given by*

$$T_{CES}(G') = T_{CES}(G) \cup \{x\ e \Rightarrow G^{*}x\ c(a)(x \in E^{*})\}.$$

(Proof is included in Appendix.)

Example 4.8 (Set of CESs of an Insert Terminal Mutant). Let *G* be the 1-Reg in Fig. 1c and $G' = It(G,p3,\{(p2,p3)\},\emptyset)$ be the 1-Reg in Fig. 5b. The set of CESs is extended by event sequences that end with *p3*, such as *x1 p2 p3*, *c1 x1 p2 p3* and *c1 p1 x1 p2 p3*.

Lemma 4.3 is used to discuss the equivalence of an insert terminal mutant to the original k-Reg in Lemma 4.4.

Lemma 4.4 (Equivalence of an Insert Terminal Mutant). *G′ is not equivalent to G if and only if $d(X)\backslash d(Y) \neq \emptyset$ where*

- *$X = \{x\ e \mid S \Rightarrow_G{}^{*} x\ c(a)(x \in E^{*})\}$ and*
- *$Y = \{w \mid w \in T_{CES}(G)$ and w contains e' where $e' \in K$, $e' \neq e$ and $d(e') = d(e)\} \subseteq T_{CES}(G)$.*

(Proof is included in Appendix.)

Example 4.9 (Equivalence of an Insert Terminal Mutant). Let G be the 1-Reg in Fig. 1c. $G' = It(G,p3,\{(p2,p3)\},\emptyset)$ in Fig. 5b extends $T_{CES}(G)$ by X that contains new sequences ending with *p3*; all these sequences actually end with *x1 p2 p3*. Although there are CESs in *G* which contain a *p* event, none of these sequences ends with an *x p p* sequence. Thus, $d(X)\backslash d(Y) \neq \emptyset$, which means that there are additional decontexted event sequences in $d(T_{CES}(G'))$. Thus, $d(T_{CES}(G')) \neq d(T_{CES}(G))$ and G' is not equivalent to *G*.

The following give sufficient conditions for usefulness, determinism, and nonequivalence of an insert terminal mutant.

Theorem 4.3 (Usefulness and Determinism of an Insert Terminal Mutant). *G′ is useful, if G is useful, and G′ is deterministic, if G is deterministic and there is no $c(a) \to e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$.*

(Proof is included in Appendix.)

Example 4.10 (Usefulness and Determinism of an Insert Terminal Mutant). Since *G*, the 1-Reg in Fig. 1c, is useful, its mutant $G' = It(G,p3,\{(p2,p3)\},\emptyset)$, the 1-Reg in Fig. 5b, is also useful. Furthermore, since *G* is deterministic and no *p* event follows *p2* in *G*, *G′* is also deterministic.

Theorem 4.4 (Nonequivalence of an Insert Terminal Mutant). *G′ is not equivalent to G, if G is useful and deterministic, and there is no $c(a) \to e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$.*

(Proof is included in Appendix.)

Example 4.11 (Nonequivalence of an Insert Terminal Mutant). *G*, the 1-Reg in Fig. 1c, is useful, deterministic and no *p* event follows *p2* in *G*. Therefore, its mutant $G' = It(G,p3,\{(p2,p3)\},\emptyset)$, the 1-Reg in Fig. 5b, is not equivalent to *G*.

The strategy to select insert terminal mutants is now given.

Insert Terminal Mutant Selection. *Given a k-Reg $G = (E,B,K,C,P)$. For each $It(G,e,\{(a,e)\},\emptyset)$, 3-tuple $(e,\{(a,e)\},\emptyset)$ is selected as a mutation parameter if the following hold:*

1. *There is no $c(a) \to x\ c(x) \in P$ such that $d(x_k) = d(e_k)$.*
2. *There is no previously selected mutation parameter $(y,\{(a,y)\},\emptyset)$ such that $d(y_k) = d(e_k)$.*

Let *G* be a useful and deterministic k-Reg. By Theorem 4.3 and Theorem 4.4, mutants generated from *G* using the above strategy are useful, deterministic, and not equivalent to *G*. Furthermore, each of these mutants models a different fault located at the mutation point; that is, e_k (also $d(e_k)$) is an extra event that follows k-sequence *a* for each $It(G,e,\{(a,e)\},\emptyset)$.

The excluded insert terminal mutants are useful. However, they are either nondeterministic or model previously modeled faults. Some nondeterministic mutants do not model any extra event faults. If they do, these faults are not located at the mutation points; therefore, they are modeled using some other insert terminal mutants.

Algorithm 4. Insert Terminal Mutant Selection

```
Input: $G = (E, B, K, C, P)$—the k-Reg
Output: $M$—the set of selected insert terminal mutants
  $M = \emptyset$
  for each $a \in K$ do
    $N = \emptyset$
    for each $b \in B$ do
      if there is no $c(a) \to x\ c(x) \in P$ such that $d(x_k) = b$ and
         there is no $(y, (a, y), \emptyset) \in N$ such that $d(y_k) = b$ then
           $b'$ = a new contexted version of $b$, $e = a_2 \ldots a_k\ b'$
           $G' = G$, $M = M \cup \{It(G', e, \{(a, e)\}, \emptyset)\}$, $N = N \cup \{e\}$
      end if
    end for
  end for
```

Algorithm 4 generates all insert terminal mutants using the above strategy. Its runtime complexity is given by $O(|K||B||P|)$: (1) The number of mutants generated is bounded by $|K||B|$ because each mutant represents a different extra event fault following a k-sequence; and (2) each mutant $It(G, e, \{(a, e)\}, \emptyset)$ can be generated in $O(|P| + |B| + k) = O(|P|)$ time by checking whether there are no $c(a) \rightarrow x\, c(x) \in P$ so that $d(x_k) = d(e_k)$ and previously selected mutation parameter $(y, \{(a, y)\}, \emptyset)$ so that $d(y_k) = d(e_k)$, preparing e by copying $a_2 \ldots a_k$ to append b', and copying G to modify it.

Also, from each selected mutant, a unique test case that kills the mutant can be generated in $O(|P|)$ time by using breadth-first search to reach $a\ e$.

Example 4.12 (Insert Terminal Mutant Selection). Let G be the 1-Reg in Fig. 1c. One can only use basis terminal *p*, because *c* and *x* can follow all terminals. The only selected insert terminal mutant is $It(G, p3, \{(p2, p3)\}, \emptyset)$, because only *p2* is not followed by a *p* event.

4.3 Negative Test Case Generation from Mutants

Each mark start mutant selected using the strategy in Section 4.1 contains a different faulty 1-sequence, aiming to reveal a different extra start event fault, and each insert terminal mutant selected using the strategy in Section 4.2 contains a different faulty (k+1)-sequence, aiming to reveal a different extra event fault. Since positive test cases cannot cover such sequences, they cannot reveal extra event faults. Thus, the inserted productions in these mutants can be covered to generate FCESs covering the mentioned faulty sequences and a unique test case can be generated for each faulty sequence to obtain a reduced test set (Algorithm 5).

Algorithm 5. Achieving Faulty (k+1)-sequence Coverage

```
Input: $G_k = (E, B, K, C, P)$—the k-Reg ($k \geq 1$)
Output: X—a set of sequences covering faulty (k+1)-sequences
  $X = \emptyset, Y = \emptyset$
  for each $G' = Ms(G_k, e)$ selected as in Section 4.1 do
    $X = X \cup \{e_1\}$
  end for
  for each $G' = It(G_k, e, \{(a, e)\}, \emptyset)$ selected as in Section 4.2 do
    s = generate a sequence by covering $c(a) \rightarrow e\ c(e)$ from $G'$
    $X = X \cup T_S^{-1}(s, k)$ //see Theorem 3.1
  end for
```

The time complexity of Algorithm 5 is given by $O(|B||P| + |K||B||P|^2)$ where (1) $O(|B||P|)$ is the time complexity of iterating through all mark start mutants using the strategy in Section 4.1; a distinguishing test case is generated in *O(1)* time from each mark start mutant; and (2) $O(|K||B||P|)$ is the time complexity of iterating through all insert terminal mutants using the strategy in Section 4.2; a distinguishing test case is generated in $O(|P|)$ time from each insert terminal mutant.

Example 4.13 (A Test Set Generated Using Algorithm 5). To achieve faulty 3-sequence coverage for the 1-Reg in Fig. 1c, the 2-Reg in Fig. 3 can be used as input to Algorithm 5. The following is an example test set:

$$\{p1, x1\ p2\ p3\}.$$

5 Case Studies

Three case studies are performed over nontrivial commercial systems to validate the approach, to analyze its characteristics, and to compare the k-Reg-based testing method to random testing [31] and mutate-and-kill-based MBMT approach (which is based on the idea of generating discriminating test cases by comparing mutants against the original model) [8], [24], [25], [4].

While performing the case studies, the following are carefully considered. (1) The SUCs are not toy systems so that the results will be nontrivial. (2) The SUCs are not immensely large so that the time spent for the case study will be convenient and the process tractable. (3) The developed models display different characteristics in the sense that the number of test targets increases in various fashions as the sequence length *k* increases. This allows considering diametrically different systems. (4) Assumptions made in Section 4 remain valid.

The case studies seek to answer the following questions.

Q1. Which approach is more effective at revealing faults?
Q2. Which approach is more cost-effective?
Q3. Which approach is more efficient at fault detection?
Q4. Which approach is more effective at revealing faults that are not targeted by the approach?
Q5. How is the test execution trend associated with each approach?

5.1 Experimental Design and Parameters

To make appropriate comparisons, *test targets* [13] are defined as (k+1)-sequences and faulty (k+1)-sequences ($k = 1, 2, 3$). The test process, the test sets generated, and the data collected using these test sets are defined as follows.

5.1.1 k-Reg

Test sequences are generated by employing the *k-Reg* approach for positive testing (Algorithm 2) and negative testing (Algorithm 5). Note that by choosing *k* values appropriately, the testing cost can be adjusted to make the approach scalable for larger applications.

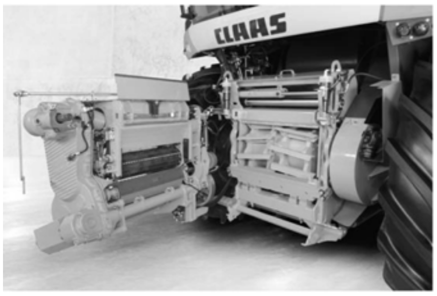
5.1.2 Mixed k-Reg (M-k-Reg)

The *mixed k-Reg* approach is used to show how *k-Reg* can be carried a 'half-step' forward if the budget is sufficient. CESs are generated from the given (k+1)-Reg for achieving (k+2)-sequence coverage, and FCESs are generated from mutants of the given k-Reg for achieving faulty (k+1)-sequence coverage.

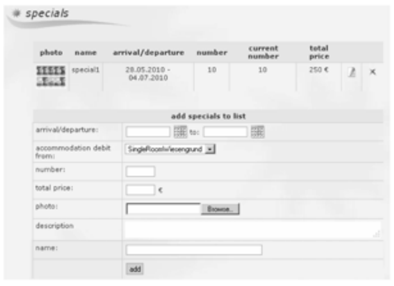
5.1.3 Random(k+1)

The *Random(k+1)* approach represents the random counterpart of the *k-Reg* and the *M-k-Reg* approaches where (k+1)-sequences and faulty (k+1)-sequences are covered using the given 1-Reg model. To collect the data for *Random(k+1)* in an appropriate manner, multiple random test sets need to be generated. Thus, *Random(k+1,maxlen)* is defined as the random testing approach where maximum length of a test sequence is bounded by *maxlen*.

In *Random(k + 1,maxlen)*, start sequences achieving (k+1)-sequence coverage and FCESs achieving faulty (k+1)-sequence coverage are generated from the given 1-Reg, adapting the approach defined in [13]. In this work, four different *maxlen* values are selected, depending on the SUC, to guarantee the coverage of the intended test targets and to



(a) `ShearBar`



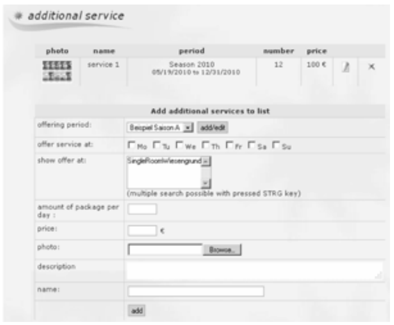
specials

photo	name	arrival/departure	number	current number	total price		
	special1	28.05.2010 - 04.07.2010	10	10	250 €		×

add specials to list

arrival/departure:	to:
accommodation debit from:	SingleRoomWiesengrund
number:	
total price:	€
photo:	Browse...
description	
name:	
	add

(b) `Specials` main interface



additional service

photo	name	period	number	price		
	service 1	Season 2010 05/19/2010 to 12/31/2010	12	100 €		×

Add additional services to list

offering period:	Beispiel Saison A add/edit
offer service at:	Mo Tu We Th Fr Sa Su
show offer at:	SingleRoomWiesengrund (multiple search possible with pressed STRG key)
amount of package per day :	
price:	€
photo:	Browse...
description	
name:	
	add

(c) `Additionals` main interface

Fig. 6. SUCs.

avoid relatively high test generation and test execution times. Furthermore, $N = 30$ random test sets are generated for each *(k+1, maxlen)* pair [11].

Consequently, the data collected using 30 random test sets are averaged to obtain the data for *Random(k+1, maxlen)*, and the data collected for *Random(k+1, maxlen)* using four different *maxlen* values are averaged to obtain the data for *Random(k+1)*.

Researches suggest that random testing performs better than a large class of testing strategies [11], [12]. Also, unlike most other random testing adaptations which do not use any information about the program or the specification [28], the adaptation in this work uses information on the test targets [13] derived from k-Reg models. Thus, this adaptation can be considered to be quite competitive.

5.1.4 Mutate and Kill (MK)

The *MK* represents the mutate-and-kill-based MBMT approach which is based on generating discriminating test cases [8], [24], [25], [4] by comparing each mutant against the original model. If the mutant is not equivalent, a set of discriminating test cases is generated using the differences between the mutant and the original model. The approach does not perform any morphology variation. Furthermore, although equivalent mutants are excluded, multiple mutants modeling the same faults are used in test generation.

Due to large numbers of possible mutants (see Table 2), test generation takes too long and test set size becomes very large. Therefore, the approach is modified to limit the executable size of the generated test set, that is, the total number of events in the test set that can be executed on the system. For each *MK*, a corresponding size is selected as the executable size of the test set generated using either *k-Reg* or *M-k-Reg*, and test cases are generated from 1-Reg mutants as long as the executable test set size is smaller than the selected size.

MK(k) is the MK-based counterpart of *k-Reg*, and *MK (M-k)* is that of *M-k-Reg*. Thus, MK-based MBMT approach is separately balanced with *k-Reg* and *M-k-Reg* in terms of the test execution effort.

5.2 Systems under Consideration

For the case studies, three nontrivial SUCs are selected from two commercial systems: (1) SFH (Self-propelled Forage Harvester) of CLAAS[1] and (2) ISELTA (Isik's System for Enterprise-Level web Centric Tourist Applications) of Isik Touristik.[2]

SFH is a farm implement that harvests forage plants. It is one of the most powerful machines used for farming, having engines generating up to 820 kW and producing an output exceeding 400 tons of silage per hour. The electronic control unit for the adjustment process of SFH shear bar (`ShearBar`) is selected (Fig. 6a) as the first case study. The `ShearBar` is controlled by signals coming from various external sources; its function is very critical for safety and financial reasons.

ISELTA is a commercial web portal for marketing tourist services. It enables travel and tourism enterprises to create their own individual search and service masks. Potential customers can then use these masks to select and book rooms and benefit from various other services. Two nontrivial facilities offered by ISELTA are selected as the second and the third case studies: `Specials` (Fig. 6b) and `Additionals` (Fig. 6c).

5.3 Models of the SUCs

A 1-Reg model is created for each SUC from the system specification, and k-Reg models for $k \geq 2$ are obtained using Algorithm 1. The properties of k-Reg models for each SUC are included in Table 1 to give some idea about the size and complexity of the models and to assure that they are not trivial. Table 1 also demonstrates that the relation between *k* and the number test targets is different for each SUC.

Table 2 gives the total numbers of mutants that are selected using the k-Reg-based approach and that can be generated using the other event-based MBMT approaches [23], [16], [42], which employ no mutant selection strategies. Since other approaches do not vary model morphology, mutants are counted using the initial system models.

Table 2 also demonstrates the reason behind using a size parameter for the MK-based approach given in Section 5.1.4 and the effectiveness of the mutant selection strategies proposed in Section 4.1 and Section 4.2. The numbers of possible mutants are quite large and the total numbers of mutants selected using different k-Reg models are ~0.40, ~2.97 and ~3.62 percent of the numbers of possible mutants.

Due to large numbers of mutants, computing the exact numbers of equivalent and multiple mutants is also not feasible, because mutants need to be compared to the original model and to the other mutants. The proposed approach avoids generation of these mutants without

1. http://www.claas.com

2. http://www.isik.de

TABLE 1
k-Reg Models

k	ShearBar		Specials		Additionals	
	k-sequences/ faulty k-sequences	k-Reg Productions	k-sequences/ faulty k-sequences	k-Reg Productions	k-sequences/ faulty k-sequences	k-Reg Productions
1	314/103	422	90/12	429	93/13	513
2	395/32,261	558	427/743	2,191	511/791	2,984
3	506/40,574	698	2,184/3,367	11,205	2,977/4,177	17,271
4	626/51,998	856	11,171/17,221	58,019	17,236/24,442	100,869

distinguishing them from each other as described in Section 4.1 and Section 4.2.

5.4 Fault Seeding

Due to large number of possible mutants for each SUC, a fixed number of event-based faults are randomly generated and seeded [52], [39], [64] to compare the testing processes in a realistic manner while gaining insight into test execution.

From an event-based MBT view, a user makes observations based on (sequences of) events. Therefore, faults can be characterized as missing event and extra event faults, because a fault in the system is observed in the form of an event that is either missing or extra at some point. m-Regs for $m = 1,2,3,4$ are used to model the faults and vary the fault domain, assuming that the faults modeled using an m-Reg generally become more subtle as m increases since a stronger coverage is required to systematically uncover them. *k-Reg*, *M-k-Reg*, *Random(k+1)*, *MK(k)*, and *MK(M-k)* aim to uncover the faults modeled using k-Reg mutants. However, it is also possible that they reveal faults modeled using m-Reg mutants for some $m \neq k$, though such faults are not targeted by them. For each case study, 50 faults are randomly seeded for each m where half of these faults are missing event faults and the other half are extra event faults. In total, 200 random faults are seeded for each SUC.

Note that using model-based faults for evaluations is actually relevant for real-world faults. There is evidence supporting the fact that a test set that detects more model-based mutants also detects more code-based mutants [6] and that a test set that detects more code-based mutants also detects more real-world faults [10]. Thus, the evidence suggests that a test set that detects more model-based mutants also detects more real-world faults.

5.5 Results of the Test Generation and Execution

Lower bounds for *maxlen* are selected to guarantee that the intended test targets can be covered and in reasonable time, and the upper bounds are selected to avoid excessive test generation and execution times. Thus, *maxlen* is limited to *60,63,67,70* for `ShearBar` and *20,30,40,50* for `Specials` and `Additionals`. Also, to collect precise data on test execution process, each test sequence is executed until a failure is observed or until its completion. Upon observing a failure, the corresponding fault is corrected and the sequence revealing this fault is re-executed. If a sequence reveals no faults and runs until completion, it is not executed again. This process continues until all test sequences are executed to completion.

Table 3 presents summarized data on test set generation and test execution processes. Table 4 outlines the data on the number of revealed faults (see Section 5.4 for m), and Fig. 7 demonstrates how the revealed number of faults changes with respect to the number of events executed.

5.6 Interpretation of the Results

Questions Q1—Q5, which are posed at the beginning of Section 5, can be now answered in order.

5.6.1 Q1: Fault Detection Effectiveness

The revealed fault numbers are rounded to the nearest integers for comparison, and Table 5 is constructed by rewriting the data for *k-Reg* and *M-k-Reg* from Table 3 with respect to *Random(k+1)* and *MK*.

Table 5 demonstrates that, in general, *k-Reg* reveals fewer faults than $Random(k+1)$, up to ~25.07 percent. However, *M-k-Reg* almost always performs better than *Random(k+1)* by revealing up to ~8.69 percent more faults. Furthermore, it shows that both *k-Reg* and *M-k-Reg* always reveal more faults than their corresponding *MK* counterparts, respectively, up to ~134.62 and ~124.39 percent.

When the change in the number of faults revealed is considered with respect to k (Table 3), *k-Reg* shows the overall fastest increasing trend for each case study, being up to ~106.06 percent faster than *Random(k+1)*. It is followed by *M-k-Reg*. Furthermore, *MK* may sometimes even show non-decreasing trends. When the increasing trends are considered,

TABLE 2
Mutant Numbers

k	ShearBar		Specials		Additionals	
	k-Reg Mutants	Event-Based Mutants	k-Reg Mutants	Event-Based Mutants	k-Reg Mutants	Event-Based Mutants
1	32,364	31,058,682	755	737,370	804	813,285
2	40,653	–	3,378	–	4,189	–
3	52,077	–	17,732	–	24,454	–
Total	125,094	31,058,682	21,865	737,370	29,447	813,285

TABLE 3
Test Generation and Test Execution Data

	Test Set	Sequence Number	Total Length	Average Length	Generation Time (s)	Events Executed	Faults Revealed	Fault Detection Rate
ShearBar	1-Reg	32,439	1,125,004	34.68	6	1,131,355	165	0.000145843
	M-1-Reg	32,465	1,126,621	34.70	6	1,133,560	174	0.000153499
	Random(2)	32,759	1,306,973	39.90	49,756	1,313,915	174.75	0.000133051
	MK(1)	30,780	1,924,440	62.52	42,927	1,129,508	103	0.000091190
	MK(M-1)	30,934	1,932,476	62.47	43,174	1,132,958	103	0.000090912
	2-Reg	40,754	1,465,701	35.96	12	1,473,026	182	0.000123555
	M-2-Reg	40,764	1,466,319	37.22	12	1,473,765	184	0.000124850
	Random(3)	41,183	1,697,887	41.23	54,756	1,705,243.25	183	0.000107362
	MK(2)	40,035	2,514,171	62.8	56,819	1,472,884	103	0.000069931
	MK(M-2)	40,035	2,514,171	62.8	56,706	1,472,884	103	0.000069931
	3-Reg	52,188	1,942,081	37.21	19	1,949,959	194	0.000099489
	M-3-Reg	52,232	1,944,064	35.97	20	1,951,997	195	0.000099898
	Random(4)	52,727	2,231,070	42.31	95,606	2,238,921	194	0.000086684
	MK(3)	52,377	3,323,226	63.45	74,683	1,947,314	108	0.000055461
	MK(M-3)	52,512	3,330,828	63.43	74,856	1,951,490	108	0.000055342
Specials	1-Reg	832	7,387	8.88	1	8,613	72	0.008359457
	M-1-Reg	1,349	16,358	12.13	3	18,407	96	0.005215407
	Random(2)	1,182	25,852	21.87	746	27,729.5	95.25	0.003631357
	MK(1)	513	13,094	25.52	12	12,116	40	0.003301420
	MK(M-1)	995	25,327	25.45	18	20,127	43	0.002136434
	2-Reg	3,972	41,548	10.46	4	43,851	122	0.002782149
	M-2-Reg	7,584	100,593	13.26	49	103,638	147	0.001418399
	Random(3)	5,563	127,968	23.00	12,211	130,738	135.25	0.001093049
	MK(2)	2,684	65,775	24.51	44	46,237	52	0.001124640
	MK(M-2)	6,268	153,337	24.46	99	106,033	69	0.000650741
	3-Reg	21,438	250,383	11.68	58	253,668	166	0.000654399
	M-3-Reg	39,878	576,675	14.46	3,928	580,495	178	0.000306635
	Random(4)	28,404	677,718	23.86	166,003	681,379.75	172.5	0.000268535
	MK(3)	16,324	395,166	24.21	272	256,265	83	0.000323883
	MK(M-3)	38,829	928,670	23.92	722	581,852	98	0.000168428
Additionals	1-Reg	910	8,154	8.96	1	9,154	65	0.007100721
	M-1-Reg	1,684	21,217	12.60	5	22,909	92	0.004015889
	Random(2)	1,315	29,462	22.40	3,967	31,191.75	86.75	0.002950497
	MK(1)	737	15,776	21.41	9	11,008	38	0.003452035
	MK(M-1)	1,519	33,982	22.37	23	22,796	41	0.001798561
	2-Reg	5,069	53,172	10.49	6	55,086	114	0.002069491
	M-2-Reg	11,019	148,652	13.49	150	151,403	139	0.000918080
	Random(3)	7,167	167,999	23.44	71,991	170,767.5	129.25	0.000801930
	MK(2)	3,803	83,448	21.94	55	57,384	57	0.000993308
	MK(M-2)	9,587	219,853	22.93	131	151,339	67	0.000442715
	3-Reg	31,284	364,059	11.64	170	367,094	161	0.000438580
	M-3-Reg	65,644	959,294	14.61	12,930	962,656	177	0.000183866
	Random(4)	41,691	1,010,872	24.25	1,034,561	1,014,664.5	172.25	0.000180284
	MK(3)	23,654	542,688	22.94	351	369,004	81	0.000219510
	MK(M-3)	62,794	1,434,617	22.85	1,087	965,881	100	0.000103532

k-Reg and *M-k-Reg* are up to ~3.2 times and ~1.2 times faster than *MK(k)* and *MK(M-k)*, respectively.

5.6.2 Q2: Cost Effectiveness

Test execution time can be measured by assuming that the execution of each event takes approximately the same amount of time on the average and taking one time unit to be the average time to execute a single event [18]. Note that using the number of executed events in this way as an indicator of the test execution effort is more realistic than using other common indicators such as the number of test cases and the total number events in the test set [61]. Thus, Table 6 is constructed using the data in Table 3 by rounding test generation times appropriately and calculating how much fewer events are executed with respect to random testing. Also, the numbers of events executed by k-Reg-based approaches are not discussed with respect to the MK-based approaches because the approaches are balanced in terms of the test execution effort as discussed in Section 5.1.4.

Table 6 shows the effects of linear-time test generation from the mutants in the k-Reg-based testing approach. In general, test generation times are much smaller for *k-Reg* and *M-k-Reg* when compared to others, up to ~99.99 percent. However, in some cases, test generation times for *M-k-Reg* are greater when compared to *MK(M-k)*, up to ~4.5 times, because *M-k-Reg* uses the corresponding (k+1)-Reg (instead of the k-Reg) for positive test generation. Also, *k-Reg* and *M-k-Reg* require, respectively, up to ~70.65 and ~33.62 percent less test execution efforts than *Random(k+1)*.

TABLE 4
Faults Revealed

	ShearBar				Specials				Additionals			
Test Set	m = 1	m = 2	m = 3	m = 4	m = 1	m = 2	m = 3	m = 4	m = 1	m = 2	m = 3	m = 4
1-Reg	50	47	35	33	50	16	6	0	50	13	2	0
M-1-Reg	50	47	39	38	50	32	12	2	50	32	8	2
Random(2)	50.00	47.25	39.70	37.50	50.00	29.62	11.29	4.20	50.00	24.43	8.79	3.52
MK(1)	28	25	26	24	25	10	2	3	25	11	2	0
MK(M-1)	28	25	26	24	26	11	2	4	26	12	3	0
2-Reg	50	50	44	38	50	50	18	4	49	50	12	3
M-2-Reg	50	50	45	39	50	50	33	14	49	50	30	10
Random(3)	50.00	50.00	43.42	39.42	50.00	50.00	23.46	11.82	49.00	50.00	18.89	10.96
MK(2)	28	25	26	24	29	15	4	4	31	20	3	3
MK(M-2)	28	25	26	24	34	22	8	5	33	25	4	5
3-Reg	50	50	50	44	49	49	50	18	48	50	50	13
M-3-Reg	50	50	50	45	49	49	50	30	48	50	50	29
Random(4)	50.00	50.00	50.00	44.08	49.00	49.00	50.00	24.45	48.00	50.00	50.00	24.09
MK(3)	28	27	28	25	36	28	12	7	35	31	8	7
MK(M-3)	28	27	28	25	40	34	16	8	38	36	15	11

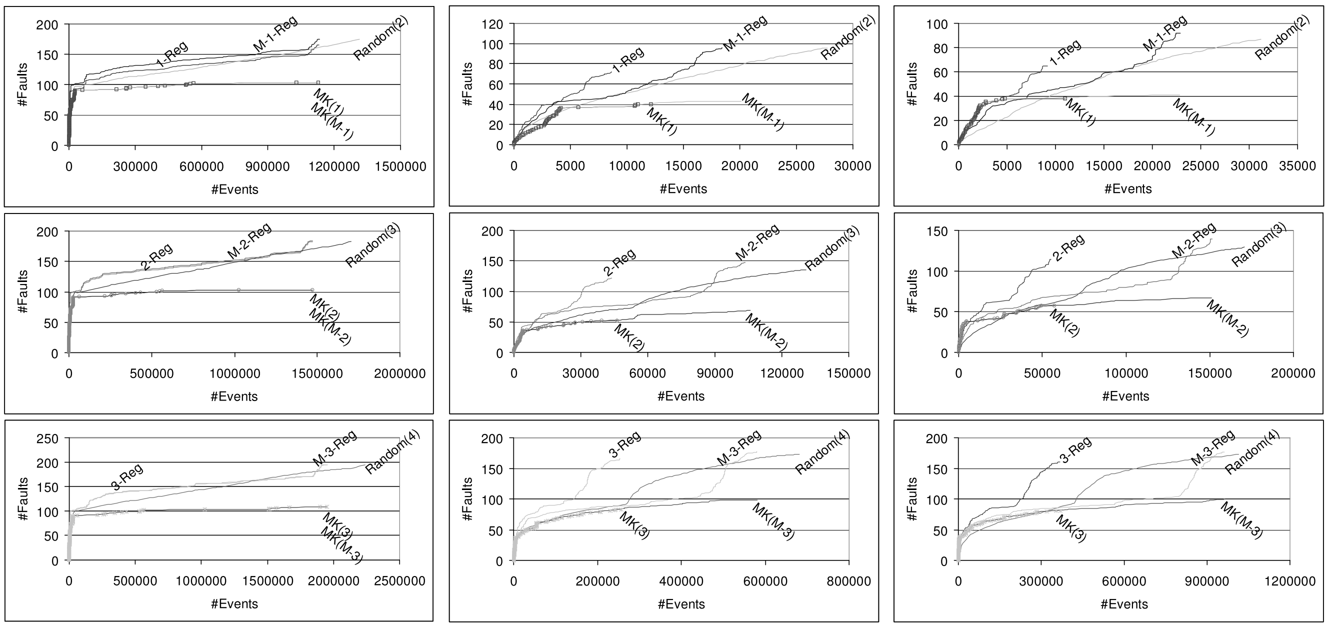
In addition, Table 3 suggests that as k increases, test generation time increases,respectively, up to ~99.99 to ~99.98 percent less for *k-Reg* and *M-k-Reg* when compared to *Random(k+1)*. Furthermore, although it generally increases, respectively, up to ~99.96 percent less for *k-Reg* and *M-k-Reg* when compared to *MK(k)* and *MK(M-k)*, the increase is sometimes greater for *M-k-Reg* when compared to *MK (M-k)*, up to 12.4 times. As for the change in test execution effort with increasing k, *k-Reg* and *M-k-Reg* show, respectively, up to~67.09 and ~17.26 percent less increase when compared to *Random(k+1)*.

5.6.3 Q3: Fault Detection Efficiency

The *fault detection rate (FDR)* (the ratio of the number of revealed faults to the number of executed events) can be used to compare fault detection efficiency. Since test execution time is measured by the number of executed events in Section 5.6.2, FDR is also formulated as the inverse of *cost per detected fault (CPF)*, that is, $FDR = 1 / CPF$.

Using Tables 3, 7 shows the differences in FDRs with respect to *Random(k+1)* and *MK*. According to Table 7, FDRs of *k-Reg* and *M-k-Reg* are always higher than *Random(k+1)*, up to ~158.06 and ~43.62 percent, respectively. Furthermore, they are also always higher than *MK(k)* and *MK(M-k)*, respectively, up to ~153.21 and ~144.12 percent.

As for the change in FDR as k increases (Table 3), all approaches show decreasing trends. *k-Reg* shows, respectively, up to ~162.35 and ~165.74 percent faster decreasing trend than *Random(k+1)* and *MK(k)*, and *M-k-Reg* shows, respectively, up to ~49.59 and ~155.57 percent faster decreasing trend than *Random(k+1)* and *MK(M-k)*. Nevertheless, as mentioned above, FDRs of *k-Reg* and *M-k-Reg* always remain greater than *Random(k+1)* and *MK*.



(a) ShearBar (b) Specials (c) Additionals

Fig. 7. Test execution curves.

TABLE 5
More Faults Revealed w.r.t. Random(k+1) and MK

		w.r.t. Random(k+1)			w.r.t. MK		
		k = 1	k = 2	k = 3	k = 1	k = 2	k = 3
ShearBar	**k-Reg**	~−5.58% (−10)	~−0.55% (−1)	same	**~60.19% (62)**	**~76.70% (79)**	**~79.63% (86)**
	M-k-Reg	~−0.43% (−1)	**~0.55% (1)**	**~0.52% (1)**	**~68.93% (71)**	**~78.64% (81)**	**~80.56% (87)**
Specials	**k-Reg**	~−24.41% (−23)	~-9.80% (−13)	~-3.79% (−7)	**~80.00% (32)**	**~134.62% (70)**	**~100.00% (83)**
	M-k-Reg	**~0.79% (1)**	**~8.69% (12)**	**~3.19% (5)**	**~123.26% (53)**	**~113.04% (78)**	**~81.63% (80)**
Additionals	**k-Reg**	~−25.07% (−22)	~−11.80% (−15)	~−6.53% (−11)	**~71.05% (27)**	**~100.00% (57)**	**~98.77% (80)**
	M-k-Reg	**~6.05% (5)**	**~7.54% (10)**	**~2.76% (5)**	**~124.39% (51)**	**~107.46% (72)**	**~77.00% (77)**

5.6.4 Q4: Effectiveness of Detecting Non-Targeted Faults

Table 4 suggests that, as k is increased, k-Reg-based testing reveals significantly more of the faults generated from an m-Reg with higher m. This is achieved by using morphologically different models to extend the set of fault models. Random testing also shows a similar trend because the test targets are derived from the k-Reg models with different k. However, such a trend is not observed for MK-based MBMT approach since it uses a single fixed model.

Using Tables 4, 8 is constructed to compare the effectiveness of the approaches at detecting faults that are not targeted by them. The percentage of more (or fewer) faults revealed by *k-Reg* and *M-k-Reg* are given with respect to $Random(k{+}1)$ and *MK* for $m = 1, 2, 3, 4$ (see Section 5.4 for m).

Table 8 shows that *k-Reg* is overall up to ~59.17 percent less effective at detecting non-targeted faults than *Random(k+1)*. On the other hand, *M-k-Reg* is always more effective than *Random(k+1)* at detecting non-targeted faults, overall up to ~14.32 percent. In addition, *k-Reg* and *M-k-Reg* are overall up to ~94.59 and ~180.00 percent more effective than *MK(k)* and *MK(M-k)*, respectively.

5.6.5 Q5: Test Execution Trends

`ShearBar`. The overall execution trends of *k-Reg* and *M-k-Reg* for `ShearBar` (Fig. 7a) are very similar to each other, especially as k increases. All the approaches reveal faults very quickly at the beginning of the test execution. Half of all the revealed faults are discovered by performing ~0.5 percent of the test execution for *k-Reg* and *M-k-Reg* and ~1 percent for $Random(k{+}1)$ and ~0.5 percent for *MK*.

For *Random(k+1)* and *MK*, the rate of change in FDR almost always decreases steadily until the end. For *k-Reg* and *M-k-Reg*, there are two points during test execution where the rate of change in the FDR shows a sudden increase. The first point resides between ~5 and ~7 percent of the test execution and the second point resides around ~95 percent.

In general, *k-Reg* and *M-k-Reg* show better FDRs than *Random(k+1)* until the end stages of their respective test execution processes where the number of faults revealed by them may, for a short while, remain up to ~4.52 percent lower than that of *Random(k+1)*. This starts at some point after ~68 to ~78 percent of the test execution is completed. After a while, the rates of change in FDRs of *k-Reg* and *M-k-Reg* increase significantly by detecting, respectively, up to ~9.70 and ~8.05 percent of the revealed faults for the last ~5 percent of the execution. In addition, *k-Reg* and *M-k-Reg* achieves better

TABLE 6
Test Generation and Test Execution Costs

		Test Generation Time (s = seconds, m = minutes, h = hours, d = days)			Fewer Events Executed w.r.t. Random(k+1)		
		k = 1	k = 2	k = 3	k = 1	k = 2	k = 3
ShearBar	**k-Reg**	**6 s**	**12 s**	**19 s**	**13.89%**	**13.62%**	**12.91%**
	M-k-Reg	**6 s**	**12 s**	**20 s**	**13.73%**	**13.57%**	**12.82%**
	Random(k+1)	13.8 h	15.2 h	26.6 h	–	–	–
	MK(k)	11.9 h	15.8 h	20.7 h	–	–	–
	MK(M-k)	12.0 h	15.8 h	20.8 h	–	–	–
Specials	**k-Reg**	**1 s**	**4 s**	**58 s**	**68.94%**	**66.46%**	**62.77%**
	M-k-Reg	**3 s**	**49 s**	1.09 h	33.62%	20.73%	14.81%
	Random(k+1)	12.5 m	3.4 h	46.1 h	–	–	–
	MK(k)	12 s	44 s	272 s	–	–	–
	MK(M-k)	18 s	99 s	12.0 m	–	–	–
Additionals	**k-Reg**	**1 s**	**6 s**	**170 s**	**70.65%**	**67.74%**	**63.82%**
	M-k-Reg	**5 s**	150 s	3.6 h	26.55%	11.34%	5.13%
	Random(k+1)	1.1 h	20.0 h	12 d	–	–	–
	MK(k)	9 s	55 s	5.9 m	–	–	–
	MK(M-k)	23 s	131 s	18.1 m	–	–	–

TABLE 7
Higher Fault Detection Rate w.r.t. Random (k+1) and MK

		w.r.t. Random(k+1)			w.r.t. MK		
		k = 1	**k = 2**	**k = 3**	**k = 1**	**k = 2**	**k = 3**
ShearBar	**k-Reg**	9.61%	15.08%	14.77%	**59.92%**	**76.76%**	**79.26%**
	M-k-Reg	**15.37%**	**16.29%**	**15.24%**	**68.87%**	**78.61%**	**80.65%**
Specials	**k-Reg**	**130.20%**	**154.53%**	**143.69%**	**153.21%**	**147.39%**	**102.04%**
	M-k-Reg	43.62%	29.77%	14.19%	**144.12%**	**117.98%**	**82.09%**
Additionals	**k-Reg**	**140.66%**	**158.06%**	**143.27%**	**105.70%**	**108.35%**	**99.81%**
	M-k-Reg	36.11%	14.48%	1.99%	**123.28%**	**107.38%**	**77.65%**

FDRs than *MK*, except for the first ~1 to ~7 percent of the test execution depending on the value of *k*, where they all show similar trends and achieve similar FDRs.

When the points where *k-Reg* and *M-k-Reg* run out of events to execute are considered as the stopping points for random testing, both *k-Reg* and *M-k-Reg* manage to detect up to ~2.48 and ~8.07 percent more faults, respectively, than *Random (k+1)*. *Random(k+1)* increases the number of revealed faults by detecting up to ~7.87 percent of the revealed faults and executing up to ~13.89 percent of all the executed events after the test execution ends for *k-Reg* and *M-k-Reg*. In addition, *k-Reg* and *M-k-Reg* detect, respectively, up to ~79.63 and ~80.56 percent more faults than their *MK* counterparts.

`Specials.` The shapes of *k-Reg*, *M-k-Reg*, *Random (k+1)* and *MK* test execution curves for Specials (Fig. 7b) are quite different from each other, except for the fact that *MK(M-k)* is an extension of *MK(k)*.

For *Random(k+1)*, the rate of change in FDR decreases as the test execution proceeds. After ~40 percent of the test execution is completed, a sudden increase in the rate of change in FDR occurs. Such increases are more frequent for *k-Reg* and *M-k-Reg*, but the most significant ones happen, respectively, around ~60 to ~70 percent and ~80 to ~85 percent of the test execution. *MK(k)* and *MK(M-k)* also show frequent increases, but they are not restricted to specific intervals. These increases become more apparent as *k* gets larger for *k-Reg*, *M-k-Reg* and *Random(k+1)* and less apparent but more frequent for *MK(k)* and *MK(M-k)*.

If the point where *k-Reg* test execution ends is considered as the stopping point for all approaches, *k-Reg* and *M-k-Reg* are, respectively, up to ~85.99 and ~8.96 percent more effective than $Random(k{+}1)$ at fault detection at the end. Furthermore, *k-Reg* and *M-k-Reg* are more effective than *MK (k)* and *MK(M-k)*, respectively, up to ~134.62 and ~46.15 percent. In this period, *k-Reg* is always more effective than $Random(k{+}1)$ at any point, *M-k-Reg* is more effective than *Random(k+1)* except until the end stages where they sometimes reach similar values. In addition, *MK* is almost always less effective than other approaches except for some short intervals (from ~45.54 to ~58.54 percent for $k = 1$, from ~8.18 to ~15.19 percent for $k = 2$, and from 0 to ~3.05 percent and from ~21.82 to ~27.13 percent for $k = 3$) where it becomes slightly more effective than $Random(k{+}1)$ for all *k* and *M-k-Reg* for $k = 2$.

Setting the end of *M-k-Reg* test execution as the stopping point, *M-k-Reg* is, respectively, up to ~30.17 and ~123.26 percent more effective than $Random(k{+}1)$ and *MK(M-k)* at fault detection at the end. The FDR of M-k-Reg becomes similar to *Random(k+1)* from ~42 to ~53 percent of test execution for $k = 1$. Also, *M-k-Reg* becomes up to ~29.66 percent less

TABLE 8
Percentage of More Faults Revealed w.r.t. Random (k+1) and MK for m = 1, 2, 3, 4 and Overall

		w.r.t. Random(k+1)			w.r.t. MK		
		k = 1	**k = 2**	**k = 3**	**k = 1**	**k = 2**	**k = 3**
ShearBar	**k-Reg**	0.00, −0.53, −11.84, −12.00 (Overall: −7.59)	0.00, 0.00, **1.34**, −3.60 (Overall: −0.63)	0.00, 0.00, 0.00, −100.00 (Overall: −0.06)	**78.57**, **88.00**, **34.62**, **37.50** (Overall: **53.33**)	**78.57**, **100.00**, **69.23**, **58.33** (Overall: **69.23**)	**78.57**, **85.19**, **78.57**, **76.00** (Overall: **80.00**)
	M-k-Reg	0.00, −0.53, −1.76, **1.33** (Overall: −0.36)	0.00, 0.00, **3.64**, −1.07 (Overall: **0.87**)	0.00, 0.00, 0.00,**2.09** (Overall: **0.64**)	**78.57**, **100.00**, **69.23**, **58.33** (Overall: **65.33**)	**78.57**, **100.00**, **73.08**, **62.50** (Overall: **71.79**)	**78.57**, **85.19**, **78.57**, **80.00** (Overall: **81.25**)
Specials	**k-Reg**	0.00, −45.98, −46.86, −100.00 (Overall: −51.23)	0.00, 0.00, −23.27, −66.16 (Overall: −15.57)	0.00, 0.00, 0.00, −26.38 (Overall: −6.27)	**100.00**, **60.00**, **200.00**, −100.00 (Overall: 46.67)	**72.41**, **233.33**, **350.00**, 0.00 (Overall: **94.59**)	**36.11**, **75.00**, **316.67**, **157.14** (Overall: **63.38**)
	M-k-Reg	0.00, **8.04**, **6.29**, −52.38 (Overall: **1.97**)	0.00, 0.00, **40.67**, 18.44 (Overall: **13.74**)	0.00, 0.00, 0.00, **22.70** (Overall: **4.53**)	**92.31**, **190.91**, **500.00**, −50.00 (Overall: **170.59**)	**47.06**, **127.27**, **312.50**, **180.00** (Overall: **106.38**)	**22.50**, **44.12**, **212.50**, **275.00** (Overall: **56.10**)
Additionals	**k-Reg**	0.00, −46.79, −77.25, −100.00 (Overall: −59.17)	0.00, 0.00, −36.47, −72.63 (Overall: −18.83)	0.00, 0.00, 0.00, −46.04 (Overall: −9.08)	**100.00**, **18.18**, 0.00, 0.00 (Overall: **15.38**)	**58.06**, **150.00**, **300.00**, 0.00 (Overall: **72.97**)	**37.14**, **61.29**, **525.00**, **85.71** (Overall: **52.05**)
	M-k-Reg	0.00, **30.98**, −8.99, −43.18 (Overall: **14.32**)	0.00, 0.00, **58.81**, −8.76 (Overall: **12.87**)	0.00, 0.00, 0.00, **20.38** (Overall: **4.02**)	**92.31**, **166.67**, **166.67**, **n/a** (Overall: **180.00**)	**48.48**, **100.00**, **650.00**, **100.33** (Overall: **111.90**)	**26.32**, **38.39**, **233.33**, **163.64** (Overall: **49.41**)

effective for $k = 2,3$ at some interval during the test execution. The length of this interval increases for larger k (from ~54 to ~86 percent for $k = 2$, and from ~43 to ~88 percent for $k = 3$). In addition, the FDR of *M-k-Reg* is always greater than *MK(M-k)* except for the very beginnings (up to the first ~5.53 percent) of the test execution where it is similar to *MK(M-k)*.

`Additionals`. The shapes of test execution curves for *k-Reg*, *M-k-Reg*, and *Random(k+1)* for `Additionals` (Fig. 7c) are relatively similar to those of `Specials`, and, as in `Specials`, *MK(M-k)* is an extension of *MK(k)*.

In general, *Random(k+1)* shows a decreasing rate of change in FDR as the test execution proceeds. However, as in `Specials`, just after ~40 percent of the test execution, the rate of change in FDR shows an increase. For *M(k)* and *MK(M-k)*, even if such increases happen, they are not very significant; whereas, for *M-k-Reg* and *k-Reg*, such sudden increases are more frequent, and they becomes more apparent as k gets larger. For *M-k-Reg*, the most significant increase happens around ~82 to ~87 percent of the test execution, and for *k-Reg* around ~55 to ~68 percent.

k-Reg and *M-k-Reg* are respectively up to ~90.80 and ~13.81 percent more effective than *Random(k+1)* at fault detection at the end, when the point where *k-Reg* test execution ends is set as the stopping point for all approaches. At this point, they are also more effective than *MK(k)* and *MK (M-k)*, respectively, up to ~100.00 and ~19.30 percent. In this period, *k-Reg* is always more effective than $Random(k+1)$ at any point, and a similar argument holds for *M-k-Reg* except for the end stages of test execution for $k = 3$ where *M-k-Reg* and *Random(k+1)* reach similar FDRs. Also, *MK* is more or equally effective as all the other approaches for the first ~34.58 percent (for $k = 1$), ~12.69 percent (for $k = 2$) and ~3.03 percent (for $k = 3$) of the test execution.

If the end of *M-k-Reg* test execution is considered as the stopping point, *M-k-Reg* is always and, respectively, up to ~24.32 and ~124.39 percent more effective than $Random(k+1)$ and *MK(M-k)* at the end. In certain intervals, *M-k-Reg* becomes less or equally effective as *Random(k+1)*. The lengths of these intervals increase with k. For $k = 1$, *M-k-Reg* becomes similar to $Random(k+1)$ from ~55 to ~90 percent of the test execution; for $k = 2$, *M-k-Reg* becomes up to ~21.74 percent less effective from ~48 to ~86 percent of the test execution; and, for $k = 3$, it becomes up to ~34.78 percent less effective from ~37 to ~96 percent of the test execution. In addition, the FDR of *M-k-Reg* is less than *MK (M-k)* for the first ~28.27 percent (for $k = 1$), ~6.80 percent (for $k = 2$) and ~2.80 percent (for $k = 3$) of the test execution, and it is greater in the rest.

5.7 A Brief Summary of the Results

The new *k-Reg* approach detects on the average,

- for `ShearBar`, ~13 percent more faults per executed event;
- for `Specials`, ~143 percent more faults per executed event; and
- for `Additionals`, ~147 percent more faults per executed event,

when compared to the random testing approach, and,

- for `ShearBar`, ~72 percent more faults per executed event;
- for `Specials`, ~134 percent more faults per executed event; and
- for `Additionals`, ~105 percent more faults per executed event,

when compared to the MK-based MBMT approach by balancing the test execution efforts.

Also, the new *M-k-Reg* approach detects on the average,

- for `ShearBar`, ~16 percent more faults per executed event;
- for `Specials`, ~29 percent more faults per executed event; and
- for `Additionals`, ~18 percent more faults per executed event,

when compared to the random testing approach, and,

- for `ShearBar`, ~76 percent more faults per executed event;
- for `Specials`, ~115 percent more faults per executed event; and
- for `Additionals`, ~103 percent more faults per executed event,

when compared to the MK-based MBMT approach by balancing the test execution efforts.

The above data suggest that k-Reg-based testing is always superior to the random testing and to the MK-based MBMT by balancing the test execution efforts, in terms of fault detection efficiency that is quantified by fault detection rate.

In addition, although morphologically different models are used, k-Reg-based testing approach decreases the mutant numbers significantly. The numbers of mutants selected using different k-Reg models ($k = 1, 2, 3$) are ~0.40 percent (for `ShearBar`), ~2.97 percent (for `Specials`) and ~3.62 percent (for `Additionals`) of the numbers of mutants required by other event-based MBMT approaches with no mutant selection strategies.

As mentioned in Section 5.2, the SUCs used in the case studies have different characteristics. The number of test targets in `ShearBar` increases linear in k; whereas, the numbers of test targets in `Specials` and `Additionals` increase exponential in k, with `Additionals` displaying an increase which is ~$1.32e^{0.1497k}$ as fast as `Specials` (see Table 1 for trends). The results suggest that the difference between *k-Reg*/*M-k-Reg*, *Random(k+1)* and *MK* is relatively less apparent for `ShearBar` when compared to `Specials` and `Additionals`.

To sum up, the relation between the number of test targets and k seems to greatly affect the difference between the results observed using different approaches.

5.8 Threats to Validity

Case studies can be applied as a comparative research strategy as used in this work. However, a case study is more of an observational method that is conducted to investigate a single entity or phenomenon. Therefore, case studies sample from the variables representing the typical situation. This makes them easier to plan but the results become difficult to generalize. Such properties make case studies prone to several threats to validity [62].

Threats to external validity. Due to the nature of case studies, different results may be obtained using different SUCs and setups. To minimize this threat, we select and use three diametrically different SUCs representing different typical situations. More precisely, since an event-based approach is used, k-sequences and faulty k-sequences of events are considered as test targets, and the approaches are compared in terms of their effectiveness and efficiency of covering these test targets and detecting faults that are intended to be revealed by these targets. Hence, the characteristics of a system are defined by the relation between *k* and the number of test targets. Different SUCs are selected and used where this relation is either (1) linear, or (2) exponential, or (3) again exponential; however, with a faster increasing trend.

Furthermore, to minimize the threat that the random testing approach is not properly adapted, the existing algorithm [13] is used and 30 test sets [11] are generated for each *Random(k+1, maxlen)* and four different *maxlen* values are selected; in total, 120 test sets are used to collect data for each *Random(k+1)*. The adaptation used in this work can be considered as an over-adaption, because it uses information on the test targets derived from k-Reg models; whereas, most random testing approaches do not use any information about the program or the specification [28].

In addition, the MK-based MBMT approach is balanced against the k-Reg-based testing approach in terms of the test execution effort as described in Section 5.1.4. This is likely to reduce the fault detection effectiveness of the method for the sake of completing the case studies in a feasible time as discussed in Sections 5.1.4 and 5.3.

Threats to internal validity. There is no prior work on which type of event-based faults are more common than the others in practice. To mitigate this threat, faults are generated and seeded randomly, avoiding any bias. To avoid a very large number of faults, a fixed number is selected, with half of the faults missing event faults and the other half with extra event faults. Also, different m-Reg models for $m = 1,2,3,4$ (see Section 5.4) are used to generate faults that are not really targeted by a specific approach and that generally become more subtle as *m* increases.

During generation of random test sets for each *Random (k+1, maxlen)*, *maxlen* values are bounded from below to guarantee that the related test targets can be covered and in reasonable time. Furthermore, an upper bound is used to avoid relatively high test generation and execution times. One can argue that the upper bound can be increased further. However, the trend observed shows that this increase would be mostly in favor of k-Reg-based testing approaches because the increase in the number of revealed faults does not seem to compensate for the increase in the test effort for greater *maxlen* values.

Threats to construct validity. For the sake of being more realistic while discussing the effectiveness at fault detection (see Section 5.6.1), the discussion was formulated as if the total numbers of faults in the SUCs were not known. Therefore, although the conclusions still apply, the calculated values would be different if the discussion were held with respect to the number of seeded faults; for example, by using the ratio of the number of revealed faults to the number of seeded faults.

Also, only the fault detection rate was used in the comparison of fault detection efficiency (see Section 5.6.3), and the effect of the test generation time was ignored. This is not a major threat because, unless the system model does not change, test sets are generated only once using a specific method. Also, even if test generation times were included, the results would be more in favor of k-Reg-based testing approaches, as suggested by the trends in Table 3.

6 Related Work

Related work can now be discussed easier in relevant categories since the approach has been introduced.

6.1 "Transformation" and "Mutation"

The grammar transformation is used in this paper for varying the morphology of a given model. This should not be confused with similar notions used in other approaches, such as model transformation [48], input/output transformation in metamorphic testing [67], and model composition [45] in aspect oriented modeling.

In model transformation, the goal is to produce a certain set of models possessing different syntax (or even semantics) and to ensure that they describe the same phenomena in a consistent way by defining relationships between these models. The grammar transformation also defines the relationship between certain types of event-based models (EBMs), that is, between a particular RG model describing a given system using k-sequences and another one describing the same system using (k+1)-sequences. However, its main purpose is to generate models of the same type with different morphological properties.

In metamorphic testing, a relation is used to reflect the changes in the input to the changes in the output so that the program can be tested, starting with a set of initial test cases, by checking the relations among several executions rather than individual outputs. In this way, no further involvement of a test oracle is needed. Hence, the way this paper varies morphology is different. Also, the proposed event-based model optionally enables to embed the test oracle into the sequence; one can decide whether a system fails or passes a test case by simply executing it [15].

In aspect-oriented modeling [45], the base model and its aspects are constructed separately. Later, these aspects are woven onto the based model; that is, the base model is transformed by applying the aspects and using certain morphisms defined between aspect elements and the base model. The goal is to simplify the design process by modularizing the crosscutting concerns; it does not aim to vary the model morphology as in this paper.

The concept of mutation is also often used in different areas of testing. For example, in metamorphic testing, it is sometimes said that a test case input is mutated to obtain another test case input. Genetic algorithms use mutation as a genetic operator (along with crossover) to produce a new generation of test cases from the existing one [49]. In this paper, mutation is used to generate fault models from an original model. Later, test generation was performed to obtain test cases that seek to reveal certain faults determined by the morphology of the model and the test generation method.

6.2 Model-Based Mutation Testing

Mutation testing is originally proposed as a code-based approach to test a given system by using systematically generated mutants, which represent faulty versions of the system, based on certain assumptions about the developer and the faults [30], [37], [1]. Basically, it can be regarded both as an evaluation technique (to assess the fault detection adequacy of a test set by its ability to detect the mutants) [3] and as a testing technique (to improve the testing process by using the mutants) [65], [38]. A major problem is the generation of equivalent mutants, that is, mutants which are equivalent to the original system. Although it is generally not possible, certain equivalent mutants can be detected [53] and specific techniques such as program slicing can reduce the effort involved in equivalent mutant detection [41]. However, as opposed to the approach proposed in this paper, the problem of generating multiple mutants that are equivalent to each other is not considered. Furthermore, only a fixed program and its mutants are utilized, limiting the set of faults especially while using it as a testing technique.

In classical sense, mutation testing is performed, respectively, on an implementation or a model to test the implementation or the model itself. In a non-classical sense, mutation testing is used to test an implementation based on its model [27], [4], [23], which is referred to as *model-based mutation testing*.

In MBMT, mutants of a model have morphological differences; however, the primary purpose is to model faults drawn from practice [27], [4], [23]. This paper systematically generates mutants over morphologically different models. This enables the extending of the set of fault models by producing mutants not necessarily considered by other approaches that are relatively closer to the line of research considered in this paper.

Amman et al. [8], and Black et al. [24], [25] make use of model-checking to check for (bounded) state equivalence between two deterministic models. In case of non-equivalence, a counterexample is obtained and used as a test case. Nondeterministic models are also used for similar purposes [54], [26], [34].

Aichernig [4] develops a theory based on the notion of refinement, which is applied using different types of abstractions in practice [63], [5], some of which may contain nondeterminism. The idea is similar to the above. However, instead of an equivalence check, a refinement check is used for conformance. In this way, mutants that are not equivalent but conform to the original model are also discarded. Improvements to the refinement checking are performed [7].

Belli et al. [23] also use event-based models to adapt MBMT. Basic mutation operators are defined and coverage-based test generation is performed. The proposed concepts are also refined or extended in different ways [16], [20], [21], [59], [42]. In these works, equivalent mutants are not really excluded and used in coverage-based test generation to populate the test set. Also, although some works [23], [16], [42] adopt the transformation defined by Belli and Budnik [22], it is used as an intermediate step for test generation. The emerging abstraction is not exploited for the purpose of extending the set of possible fault models. Furthermore, the use of regular grammars and the mutant selection strategies for testing are considered in our previous works [16], [17]. However, exploiting model morphology, linear-time test generation from mutants, and a full-fledged approach and its detailed evaluation are not discussed.

The approach proposed in this paper does not operate using a fixed model and, thus, is not limited to a fixed set of fault models. A transformation to vary model morphology is outlined for the purpose of extending the set of fault models and generating test sets achieving different coverage to reveal additional faults. In comparison to [51], [9], the proposed mutation operators are more suitable for event-based testing and for generating mutants that contain a small number of faults. Furthermore, with the help of the mutant generation strategies devised by exploiting the simple semantics of the model, it can be guaranteed that, when a mutant is selected, the fault is located at the mutation point. Thus, in contrast to works related to [8], [4], there is no need to compare the original model to the mutant to check for (bounded) equivalence or conformance or to generate a test case that kills the mutant; one can simply use the mutation operation to do so. Furthermore, generation of equivalent mutants and multiple mutants modeling the same faults can be avoided. This helps to reduce the number of mutants significantly and eliminate masked negative test cases when compared to [23], [16], [42].

6.3 Grammars in Testing

Software testing practice contains a substantial amount of work based on grammars for generating well-formed inputs [56], [47], testing interpreters [58], and, in general, testing software termed as *grammarware*, such as compilers, debuggers, code generators, and documentation generators [44] using different grammar-based formalisms, for instance, attribute grammars [55] and graphs grammars [32]. Their use in modeling behavioral aspects of software systems has been rare because models such as FSA are preferred due to their state-based nature. Therefore, grammar-based testing generally refers to the use of grammar-based formalisms for testing grammarware.

In this respect, the approach in this paper contains similarities with an existing approach [51], [9] that also uses grammar-based models and mutation operators. However, the present paper avoids the use of *nonterminal* and *terminal duplication*, *deletion*, and *replacement* operators introduced by Offutt, Ammann, and Liu. First, all of the operators, except nonterminal duplication, can be realized by using the combinations of the event-based operators. Second, and more critically, nonterminal duplication is not type-preserving. Consequently, if this operator is applied to an RG, the mutant becomes a CFG; that is, the type of the original model is injured. This has severe impacts on decidability features relative to the undecidability of the equivalence of generated CFG-type mutants. This drawback is exemplified using the regular grammar in Fig. 8a that is drawn from an example used by Ammann and Offutt [9], and slightly, nevertheless equivalently, reformatted for saving space. The nonterminal duplication mutant in Fig. 8b is a CFG.

7 Conclusion and Future Work

This paper proposes an event- and model-based mutation testing approach that systematically varies a given grammar-based model to generate morphologically different models. This enables the generation of test cases covering longer event sequences. The major novelties are as follows.

(a) Regular grammar.	(b) Context-free grammar.
S $\rightarrow$ deposit ACC0 | debit ACC0	S $\rightarrow$ deposit ACC0 | debit ACC0
ACC0 $\rightarrow$ digit ACC1	ACC0 $\rightarrow$ digit ACC1
ACC1 $\rightarrow$ digit ACC2	ACC1 $\rightarrow$ digit ACC2
ACC3 $\rightarrow$ digit AMM0	ACC3 $\rightarrow$ digit AMM0
AMM0 $\rightarrow$ \$ AMM1	AMM0 $\rightarrow$ \$ AMM1
AMM1 $\rightarrow$ digit AMM2	AMM1 $\rightarrow$ digit **AMM2 AMM2**
AMM2 $\rightarrow$ digit AMM2	AMM2 $\rightarrow$ digit AMM2
AMM2 $\rightarrow$. AMM3	AMM2 $\rightarrow$. AMM3
AMM3 $\rightarrow$ digit AMM4	AMM3 $\rightarrow$ digit AMM4
AMM4 $\rightarrow$ digit ACT | digit S	AMM4 $\rightarrow$ digit ACT | digit S
ACT $\rightarrow$ ε	ACT $\rightarrow$ ε

Fig. 8. A regular grammar and its nonterminal duplication mutant (drawn from [9]).

- Equivalent mutants and multiple mutants modeling the same faults are excluded.
- Any mutant can be killed by a unique, dedicated test case that is generated in linear time; a comparison against the original model is not necessary.
- The associated set of fault models can systematically be extended to consider different, subtle faults.

These benefits enable to comply with quality and budgetary requirements and are accomplished by a series of nontrivial steps. First, a grammar model called *k-sequence right regular grammar ($k-Reg$) ($k \geq 1$)* is introduced to represent the relation between events sequences of length *k* (*k-sequences*) and single events. Second, a grammar transformation is defined to vary *k* for generating morphologically different models and generating corresponding test cases covering (k+1)-sequences. Third, appropriate event-based mutation operators are defined to extend the set of fault models and develop efficient mutant and test selection strategies to increase the efficiency of the test process. To the authors' knowledge, no other approach combines these advantages.

The characteristics of the approach are analyzed and a comparison against random testing is performed over three case studies based on industrial and commercial applications with different domains. An alternative of the approach is derived to perform further improvements: mixed k-Reg. The results are summarized as follows.

- *k-Reg* detected 13 to 147 percent more faults per executed event than the random testing, and 72 to 134 percent more faults per executed event than the MK-based MBMT approach.
- *M-k-Reg* detected 16 to 29 percent more faults per executed event than the random testing, and 76 to 115 percent more faults per executed event than the MK-based MBMT approach.
- The numbers of required mutants were also greatly reduced while extending the set of fault models. The number of mutants selected using different k-Reg models ($k = 1,2,3$) were 0.40 to 3.62 percent of the numbers of mutants required by the other event-based MBMT approaches which uses no mutant selection strategies.

The authors hope that they have been able to demonstrate the scalability of the approach not only through the case studies but also through the adjustability of the project budgetary costs by varying *k*.

Future work is planned to refine the approach for testing systems in which the assumptions are restrictive (Section 4). Accordingly, different traits for modeling, such as hierarchy and communication, or utilizing the semantics of specific types of systems are to be considered. The idea of varying model morphology will be applied to models having richer semantics by generalization of grammar transformation. Also, the properties of the mutation operators should be analyzed to consider usefulness preservation for constructing additional techniques for mutant selection and test generation.

APPENDIX

Appendix is available as supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments and questions to improve the paper. They also appreciate the comments of Dr. Aichernig while adapting the *mutate-and-kill-based* model-based mutation testing approach based on the idea of generating discriminating test cases [8], [24], [25], [4], which is compared with the approach proposed in this paper. Mutlu Beyazıt is the corresponding author; author names are ordered alphabetically.

REFERENCES

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis," Tech. Rep., GIT-ICS-70/08, Georgia Institute of Technology, Sep. 1979.

[2] K. Adamopoulos, M. Harman, and R. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Proc. AAAI Genetic Evolutionary Comput. Conf.*, Jun. 2004, vol. 3103, pp. 1338–1349.

[3] H. Agrawal, R. A. DeMillo, R. Hathaway, Wm. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford, "Design of mutant operators for the c programming language," Tech. Rep., Softw. Eng. Res. Center SERC-TR-41-P, Purdue Univ., West Lafayette, IN, USA, Mar. 1989.

[4] B. K. Aichernig, "Mutation testing in the refinement calculus," *Formal Aspects Comput. J.*, vol. 15, no. 2/3, pp. 280–295, Nov. 2003.

[5] B. K. Aichernig, H. Brandl, E. Jöbstl, and W. Krenn, "Model-based mutation testing of hybrid systems," in *Proc. 8th Int. Conf. Formal Methods Components Objects*,2010, pp. 228–249.

[6] B. K. Aichernig, H. Brandl, E. Jöbstl, and W. Krenn, "Efficient mutation killers in action," in *Proc. IEEE 4th Int. Conf. Softw. Testing*, Mar. 2011, pp. 120–129.

[7] B. K. Aichernig and E. Jöbstl, "Efficient refinement checking for model-based mutation testing," in *Proc. 12th Int. Conf. Quality Softw.*, Aug. 2012, pp. 21–30.

[8] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *Proc. 2nd IEEE Int. Conf. Formal Eng. Methods*, 1998, pp. 46–54.

[9] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, U.K.: Cambridge Univ. Press, 2008.

[10] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng.*, May 2005, pp. 402–411.

[11] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 1–10.

[12] A. Arcuri and L. Briand, "Formal analysis of the probability of interaction fault detection using random testing," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1088–1099, Sept./Oct. 2012.

[13] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 258–277, Mar./Apr. 2012.

[14] B. Beizer, *Software Testing Techniques*. New York, NY, USA: Van Nostrand, 1990.

[15] F. Belli, "Finite-state testing and analysis of graphical user interfaces," in *Proc. 12th Int. Symp. Softw. Rel. Eng.*, Nov. 2001, pp. 34–43.

[16] F. Belli and M. Beyazıt, "A formal framework for mutation testing," in *Proc. 4th Int. Conf. Secure Softw. Integr. Rel. Improvement*, Jun. 2010, pp. 121–130.

[17] F. Belli, and M. Beyazıt, "Using regular grammars for event-based testing," in *Proc. 18th Int. Conf. Implementation Appl. Automata*, Jul. 2013, pp. 48–59.

[18] F. Belli, M. Beyazıt, and N. Güler, "Event-oriented, model-based GUI testing and reliability assessment—Approach and case study," *Adv. Comput.*, vol. 85, pp. 277–326, 2012.

[19] F. Belli, M. Beyazıt, and A. Memon, "Testing is an event-centric activity," in *Proc. 6th Int. Conf. Softw. Security Rel.*, Jun. 2012, pp. 198–206.

[20] F.Belli, M. Beyazıt, T. Takagi, and Z. Furukawa, "Mutation testing of 'go-back' functions based on pushdown automata," in *Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation*, Mar. 2011, pp. 249–258.

[21] F. Belli, M. Beyazıt, T. Takagi, and Z. Furukawa, "Model-based mutation testing using pushdown automata," *IEICE Trans. Inf. Syst.*, vol. E95-D, no. 9, pp. 2211–2218, Sep. 2012.

[22] F. Belli and C. J. Budnik, "Minimal spanning set for coverage testing of interactive systems," in *Proc. 1st Int. Colloq. Theoretical Aspects Comput.*, Sep. 2004, pp. 220–234.

[23] F. Belli, C. J. Budnik, and W. E. Wong, "Basic operations for generating behavioral mutants," in *Proc. 2nd Workshop Mutation Anal.*, Nov. 2006, pp. 9–18.

[24] P. E. Black V. Okun, and Y. Yesha, "Mutation operators for specifications," in *Proc. 15th IEEE Int. Conf. Autom. Softw. Eng.*, 2000, pp. 81–88.

[25] P. E. Black, V. Okun, and Y. Yesha, "Mutation of model checker specifications for test generation and evaluation," *Mutation Testing for the New Century, Kluwer International Series on Advances in Database Systems*, vol. 24., W. E. Wong, Ed. Norwell, MA, USA: Kluwer, 2001, pp. 14–20.

[26] S. Boroday, A. Petrenko, and R. Groz, "Can a model checker generate tests for non-deterministic systems?" *Electron. Notes Theoretical Comput. Sci.*, vol. 190, no. 2, pp. 3–19, Aug. 2007.

[27] T. A. Budd and A. S. Gopal, "Program testing by specification mutation," *Comput. Languages*, vol. 10, no. 1, pp. 63–73, 1985.

[28] T. Y. Chen and Y. T. Yu, "On the expected number of failures detected by subdomain testing and random testing," *IEEE Trans. Softw. Eng.*, vol. 22, no. 2, pp. 109–119, Feb. 1996.

[29] N. Chomsky, "Three models for the description of language," *IRE Trans. Inf. Theory*, vol. 2, no. 3, pp. 113–124, Sep. 1956.

[30] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Comput.*, vol. 11, no. 4, pp. 34–41, Apr. 1978.

[31] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 438–444, Jul. 1984.

[32] H. Ehrig, H. Kreowski, U. Montanari, and G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 1–3. Singapore: World Scientific, 1996.

[33] M. Felleisen, "On the expressive power of programming languages," *Sci. Comput. Program.*, vol. 17, no. 1–3, pp. 35–75, Dec. 1991.

[34] G. Fraser and F. Wotawa, "Nondeterministic testing with linear model-checker counterexamples," in *Proc. 7th Int. Conf. Quality Softw.*, Oct. 2007, pp. 107–116.

[35] V. K. Garg and M. T. Ragunath, "Concurrent regular expressions and their relationship to Petri nets," *Theoretical Comput. Sci.*, vol. 96, no. 2, pp. 285–304, Apr. 1992.

[36] B. J.M. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Proc. Int. Conf. Softw. Testing, Verification Validation Workshops*, Apr. 2009, pp. 192–199.

[37] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 279–290, Jul. 1977.

[38] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 212–222.

[39] M. J. Harrold, A. J. Offutt, and K. Tewary, "An approach to fault modeling and fault seeding using the program dependence graph," *J. Syst. Softw.*, vol. 36, no. 3, pp. 273–295, Mar. 1997.

[40] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J.H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surveys*, vol. 41, no. 2, p. 76, Feb. 2009.

[41] R. M. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Softw. Testing, Verification Rel.*, vol. 9, no. 4, pp. 233–262, 1999.

[42] A. Hollmann, "Model-based mutation testing for test generation and adequacy analysis," Ph.D. dissertation, Univ. Paderborn, Paderborn, Germany, 2011.

[43] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2006.

[44] P. Klint, R. Lämmel, and C. Verhoef, "Toward an engineering discipline for grammarware," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 3, pp. 331–380, Jul. 2005.

[45] M. E. Kramer, J. Klein, J. R.H. Steel, B. Morin, J. Kienzle, O. Barais, and J.-M. Jézéquel, "On the formalisation of GeKo: A generic aspect models weaver," Tech. Rep. UL-CHAPTER-2012-280, SNT, Univ. Luxembourg, Walferdange, Luxembourg, 2012.

[46] B. A. Malloy and J. F. Power, "A top-down presentation of Purdom's sentence-generation algorithm," Tech. Rep. NUIM-CS-TR-2005-04, Nat. Univ. Ireland, Galway, Ireland, 2005.

[47] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Softw.*, vol. 7, no. 4, pp. 50–55, Jul. 1990.

[48] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theoretical Comput. Sci.*, vol. 152, pp. 125–142, Mar. 2006.

[49] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1996.

[50] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Trans. Softw. Eng.*, vol. 14, no. 6, pp. 868–874, Jun. 1988.

[51] A. J. Offutt, P. Ammann, and L. Liu, "Mutation testing implements grammar-based testing," in *Proc. 2nd Workshop Mutation Anal.*, Nov. 2006, pp. 12–21.

[52] A. J. Offutt and J. H. Hayes, "A semantic model of program faults," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 1996, pp. 195–200.

[53] A. J. Offutt, and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Testing, Verification Rel.*, vol. 7, no. 3, pp. 165–192, 1997.

[54] V. Okun, P. E. Black, and Y. Yesha, "Testing with model checker: Insuring fault visibility," in *Proc. WSEAS Int. Conf. Syst. Sci., Appl. Math. Comput. Sci. Power Eng. Syst.*, Oct. 2002, pp. 1351–1356.

[55] J. Paakki, "Attribute grammar paradigms—A high-level methodology in language implementation," *ACM Comput. Surveys*, vol. 27, no. 2, pp. 196–255, Jun. 1995.

[56] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Math.*, vol. 12, no. 3, pp. 366–375, Sep. 1972.

[57] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual*, 2nd ed. Upper Saddle River, NJ, USA: Pearson Higher Educ., 2004.

[58] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *Proc. 2nd Conf. Domain-Specific Languages*, Oct. 1999, pp. 1–13.

[59] T. Takagi, R. Takata, Z. Furukawa, F. Belli, and M. Beyazıt, "Metrics for model-based mutation testing based on place/transition nets," in *Proc. Joint Conf. 21st Int. Workshop Softw. Meas. 6th Int. Conf. Softw. Process Product Meas. - Fast Abstracts*, Nov. 2011, pp. 7–10.

[60] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Mateo, CA, USA: Morgan Kaufmann 2006.

[61] A. Wood, "Software reliability growth models," Tech. Rep. 96-1, Tandem Computers Inc. - Corporate Information Center, Sep. 1996.

[62] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer, 2000.

[63] M. Weiglhofer, B. Aichernig, and F. Wotawa, "Fault-based conformance testing in practice," *Int. J. Softw. Informat.*, vol. 3, no. 2/3, pp. 375–411, Jun.–Sep. 2009.

[64] Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 1–35, Nov. 2008.

[65] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.

[66] L. Zheng and D. Wu, "A sentence generation algorithm for testing grammars," in *Proc. 33rd Int. Comput. Softw. Appl. Conf.*, Jul. 2009, vol. 1, pp. 130–135.

[67] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen, "Metamorphic testing and its applications," Tech. Rep. TR-2004-12, Univ. Hong Kong, Hong Kong, 2004.

[68] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *ACM Comput. Surveys*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

Fevzi Belli received the PhD degree in formal methods for self-correction features in sequential systems in 1978 and the "Habilitation" (German postdoctoral degree) in software engineering in 1986 from Berlin Technical University. Simultaneously, he spent several years as a software engineer in Munich, writing programs to test other programs. In 1983, he received a professorship at the University of Applied Sciences in Bremerhaven; in 1989, he moved to the University of Paderborn. Since 2014, he has been a full professor at Izmir Institute of Technology. Prior to his appointment at the University of Paderborn, he was also, for many years, a faculty member of the University of Maryland, College Park, European Division. During 2002 and 2003, he was the founding chair at the Computer Science Department, University of Economics in Izmir, Turkey. He has an interest and experience in software reliability/fault tolerance, model-based testing, and test automation.

Mutlu Beyazıt received the BSc degree in computer engineering and the MSc degree in computer software from İzmir Institute of Technology in 2005 and 2008, respectively. In 2014, he received the PhD degree from the University of Paderborn. He was a research assistant at İzmir Institute of Technology from 2005 to 2008 and at the University of Paderborn from 2009 to 2013. Since 2014, he has been with Yaşar University. His current interests include model-based testing.