Exploiting Model Morphology for Event-Based Testing

Fevzi Belli, Mutlu Beyazit

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 $k \in f$ 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—Model-based testing, grammar-based testing, (model) morphology, mutant selection, test generation

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 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% 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 subsections explain usage of formal grammars for modeling, mutant generation, and grammar transformation for varying the model morphology. Novelties are exemplified in Subsection 2.4 and an overview of the approach is given in the last subsection.

### Example 2.1 (Running Example).

Consider three events

- \( c \): copy,
- \( x \): cut, and
- \( p \): 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 \(p\).

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 \(\text{follows}\) relation between the events is visualized by arcs. Pseudo-events \([i]\) and \([j]\) 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 contexts in which they reside, leading to \(\text{contexted events}\), such as \([c1, x1, p1, p2]\) (Fig. 1b). Their counterparts, \(\text{basis events}\), 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 \(\text{follows}\) relation between single events. This modification needs to be generalized by analyzing occurrences of single events with respect to event sequences of length \(k \geq 1\) (\(k\)-\(\text{sequences}\)) for systematic extension of event-based fault modeling.

Grammars are suitable for representing event-based abstractions based on \(k\)-\(\text{sequences}\). They allow multiple occurrences of events in \(\text{productions}\), which enable to represent the \(\text{follows}\) relation between \(k\)-\(\text{sequences}\) 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 \(\text{(contexted) events}\), a set of \(\text{basis events}\), a set of \(k\)-\(\text{sequences (terminals)}\), a set of \(\text{contexts (nonterminals)}\) including a \(\text{start context}\) and a set of \(\text{productions}\). A context relation determines the right unique context of a \(k\)-\(\text{sequence}\) 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 Fig. 1b, Fig. 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-\(\text{sequence}\).
- \(S \rightarrow a\) \(c(a)\) means that \(a\) is a start event.
- \(c(a) \rightarrow e\) means that \(a\) is a finish event.

Also, \(c(a)\) denotes the \(\text{(right)}\) 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-\(\text{sequences}\), and the nonterminals are contexts. Therefore, this model is called “1-\(\text{Reg}^\ast\)”.

### 2.2 Generating Mutants

The new approach refines the elementary mutation operators \textit{insertion} and \textit{omission} [23] to modify sequences of events by also considering the \textit{start} and \textit{finish} events. The iterative and combinatorial deployment of these operations enables further mutation operators such as \textit{duplication}, \textit{deletion}, or \textit{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-\(\text{Reg}^\ast\): it
The introduced event-based grammar model enables the generation of morphologically different models by a transformation to vary morphology.

### 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 $c(a c) \rightarrow e b c(e b)$ means that $b$ follows $a e$ and $a e b$ is a 3-sequence.

#### 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

\[ p1 \text{ is missing after } c1; \]

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].

Each selected mutant has the following properties. (1) It does not violate the type-$3$ness 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 $x p$.

**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 5.3).
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 (k-Reg) (integer k ≥ 1) is a quintuple G = (E, B, K, C, P) where:

- E is a finite set of events (or contextual events).
- B is a finite set of basis events, which is the set of all visible events under consideration. For e ∈ E, d(e) ∈ B is the corresponding basis event (the noncontexted version of e), and d(.) is the decontexting function.
- K ⊆ E is a finite set of k-sequences (or terminals). For r ∈ K, r = r₁...rₖ and d(r) = d(r₁)...d(rₖ) ∈ B^k is the corresponding basis k-sequence.
- C is a finite set of contexts (or nonterminals) where S ∈ C is the start context (or start symbol).
- P is a finite set of productions of the form Q → e or Q → r c(r)

where Q ∈ C is a context, r ∈ K is a k-sequence, c(r) ∈ C \ {S} is the unique context of r, and e is the empty string. If k ≥ 2, then for each c(q) → r c(r) ∈ P where q = q₁...qₖ ∈ K and r = r₁...rₖ ∈ K,

q₂...qₖ = r₁...rₖ-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) → r c(r) ∈ P, r₁ follows q in the system modeled by grammar G; that is, q r₁ is a (k+1)-sequence in the system. Also, r is a start k-sequence for each S → r c(r) ∈ P, and q is a finish k-sequence for each c(q) → e ∈ 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 l and l. Arcs of the form “(l, r)”, “(r, l)”, and “(q, r)” correspond to the productions of the form “S → r c(r)”, “c(r) → l”, and “c(q) → r c(r)”, respectively.

Productions of a k-Reg are used to derive strings. A derivation, denoted by γ, is a sequence of derivation steps, each of which is of the form xQy → xRy where x, y ∈ (C∪K) and Q → R ∈ P (⇒ and ⇒ 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 ⇒* w (w ∈ 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 = s₁ s₂ ... sₙₙ, the corresponding basis event sequence of s is d(s) = d(s₁) d(s₂) ... d(sₙ) where d(α) = α. Furthermore, the corresponding set of basis event sequences of a set of event sequences X is d(X) = {d(s) | s ∈ X}.

Example 3.2 (Decontexted Event Sequences). Consider the 1-Reg in Fig. 1c. For event sequence s = c₁ x₁ p₂ c₁ 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 ⇒ s Q (Q ∈ C) or Q ⇒ s Q (Q ∈ 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 {p₁ p₂, 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 ≥ 1, so that s ∈ 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) → p₁ x₁ c(p₁ x₁), c(p₁ x₁) → x₁ x₁ p₂ c₁ and c(x₁ p₂) → p₂ c₁ (c₁ p₂).

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 if it is a Tᵥ(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 e if e ∈ L(G). Tᵥ(G) = L(G) ⊆ Tᵥ(G) denotes the set of all CESs.
- An event sequence is a negative test case, if the first
Example 3.5 (Test Cases of a 1-Reg). For the 1-Reg in Fig. 1c:
- \{c_1, x_1 x_1, c_1 p_1 p_1 x_1\} is a set of positive test cases, and \{x_1 p_2, x_1 x_1 p_2, c_1 p_1 p_1 p_1\} is a set of CESs.
- \{p_1, x_1 p_2 p_1 c_1, c_1 x_1 p_2 p_2\} is a set of negative test cases, and \{x_1 p_2 p_2, c_1 x_1 p_2 p_2\} is a set of FCEs.

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 \(S \Rightarrow zxy \Rightarrow w\) 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.

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 \(Q \rightarrow q c(q) \in P\) and \(Q \rightarrow r c(r) \in P\) such that \(r \neq q\) and \(d(r) = d(q)\).

Example 3.6 (A Useful and a Nonuseful 1-Reg). k-Regs in Fig. 1c and Fig. 3 are all useful. To obtain a non-useful 1-Reg from Fig. 1c, one can remove \(c(p_1) \rightarrow e\) and \(c(p_2) \rightarrow e\). The resulting grammar does not have any finish events. Therefore, \(T_{CES}(G)\) is empty, but the follows relation is still described correctly.

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

Definition 3.6 (Determinism). A k-Reg G = (E, B, K, C, P) is deterministic, if for each \(Q \in C\), there are no two productions \(Q \rightarrow q c(q) \in P\) and \(Q \rightarrow r c(r) \in P\) 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 \(c(p_1) \rightarrow p_2 c(p_2)\) in Fig. 1c is non-deterministic. Positive test cases \(s = c_1 p_2 c_1\) and \(t = c_1 p_1 c_1\) are redundant because \(d(s) = d(t)\).

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 \(c(q) \rightarrow r_1 r_2 c(r_1 r_2)\) in Fig. 3, \(q_2 = r_1\).

Thus, each such production can be obtained by using \(c(q) \rightarrow q_2 c(q)\) and \(c(q) \rightarrow r_2 c(r_2)\) in Fig. 1c.

Each \(S \rightarrow r_1 r_2 c(r_1 r_2)\) in Fig. 3 can be obtained by using \(S \rightarrow r_1 c(r)\) and \(c(r) \rightarrow r_2 c(r_2)\) in Fig. 1c.

Each \(c(r_1 r_2) \rightarrow c\) in Fig. 3 can be obtained by using \(c(r_1) \rightarrow r_2 c(r_2)\) and \(c(r) \rightarrow c\) in Fig. 1c.

Definition 3.6 (k-Reg Transformation). Given a 1-Reg \(G_1 = (E, B, C, K, P_1)\):
- The corresponding 1-Reg of \(G_1\) is itself: \(G_1\).
- Let \(G_k = (E, B, K, C, P_k)\) be the corresponding k-Reg of \(G_1\). The corresponding (k+1)-Reg of \(G_k\) (or \(G_0\)) is \(G_{k+1} = (E, B, K_{k+1}, C_{k+1}, P_{k+1})\) where:
  - \(K_{k+1} = \{q_1, \ldots, q_k, r_1\} c(q) \rightarrow r c(r) \in P_1\) 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.
  - \(P_{k+1} = \{S \rightarrow r c(r)\} \cup c(q) \rightarrow r c(r) \in P_1\) and \(c(r_1) \rightarrow e \in P_1 \cup (c(q) \rightarrow r c(r) \in P_1)\), \(c(r_1) \rightarrow e \in P_1 \cup (c(q) \rightarrow r c(r) \in P_1)\), \(c(r) \rightarrow c(r_1) \in P_1\) and \(c(r_1) \rightarrow e \in P_1\) is the set of productions.

Based on Definition 3.6, Algorithm 1 performs k-Reg transformation. It runs in \(O(k|P_1|P_1|) = O(k|P_1|^2)\) worst case time because (1) \(|P_1| = O(|P_1|^2);\) (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 \(G_k\) 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, \(|P_1|\) is generally much smaller than \(|P_1|^2\), and \(k\) is almost always bounded. Hence, transformation can be performed quite fast.

Algorithm 1. k-Reg Transformation

Input: \(G_1 = (E, B, K, C, P_1)\) - the corresponding k-Reg of \(G_1\)
Output: \(G_{k+1} = (E, B, K_{k+1}, C_{k+1}, P_{k+1})\) - the corresponding (k+1)-Reg

\[\begin{align*}
K_{k+1} &= \{q_1, \ldots, q_k, r_1\} \\
C_{k+1} &= \{c(r) | r \in K_{k+1}\} \\
P_{k+1} &= \{S \rightarrow r c(r)\} \cup c(q) \rightarrow r c(r) \in P_1\} \\
&\text{for each } Q \in C \text{ do} \\
&\text{if } Q = c(q) \text{ then } c_{k+1} = c_{k+1} \cup \{c(q) \rightarrow r c(r)\} \\
&\text{else if } Q = S \text{ then } P_{k+1} = P_{k+1} \cup \{S \rightarrow r c(r)\} \\
&\text{else if } Q = c(q) \text{ then } P_{k+1} = P_{k+1} \cup \{c(q) \rightarrow r c(r)\} \\
&\text{endif} \\
&\text{endif} \\
&\text{endfor} \\
&\text{endfor}
\end{align*}\]

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×m)-sequence \( s = u^1 \ldots u^m \) where \( k \geq 1 \), \( m \geq 1 \) and \( u^i = u_{i1} \ldots \)
\( w_i \) for \( i = 1, \ldots, m \). Inverse sequence transformation of \( s \) based on integer \( k \) is defined as a 
(k+m-1)-sequence \( T_{k}^{-1}(s, k) = u^1 u_{2}^{1} \ldots u_{m}^{1} \)
where \( u^1 = s_1 \ldots s_k \) and each \( w_i = s_{ik} \) 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 \( G_v \) and its corresponding k-Reg \( G_k \) where \( k \geq 2 \) and \( K_k \not= \emptyset \). If \( s \) is an m-derived sequence in \( G_v \),
\( t = T_{k}^{-1}(s, k) \) is a (k+m-1)-derived sequence in \( G_t \).

(Proof is included in Appendix.)

Example 3.9 (Sequence Transformation). \( T_{5}^{-1}(s, 2) = c1 \ c1 \ x1 \ x1 \ p2 \) is a 5-derived sequence in the 1-Reg in Fig. 1c for 4-derived sequence \( s = c1 \ c1 \ x1 \ x1 \ x1 \ x1 \ p2 \) 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
\( X = X \cup T_{k}^{-1}(s, k) \) //See Theorem 3.1
endfor

The time complexity of Algorithm 2 is given by
\( O(C(G_k |E|, |P|) + C_r(G_k |E|, |P|)) = O(C(G_k |E|, |P|)) \)
(1) \( C_r(G_k |E|, |P|) \) is the time complexity of generating a set of sequences achieving production coverage for \( G_k \) and
(2) \( C_r(G_k |E|, |P|) \) 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 \( O(|K|^1) = O(|P|^1) \) [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 x1 x1 p2 c1 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.

Algorithms that differ in the result operators ([23][51][9]) can be defined for k-Reg 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.

Definition 4.1 (Mark Start). Given a k-sequence \( e \in K \) such that \( S \rightarrow e c(e) \not\in P \), 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
The set of all CESs of a Mark Start Mutant.

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

$$CES(G') = CES(G) \cup \{e x | c(e) \Rightarrow c' x (x \in E')\}.$$ 

(Proof is included in Appendix.)

**Example 4.2 (Set of CESs 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. Since there is no CES starting with a $p$ event in $G$, $Y = \emptyset$. Thus, $d(X) \neq d(Y)$ and $d(X)$ is an extra start event for each $e \in E$. Furthermore, each of these mutants models a different fault located at the mutation point; that is, $e_i$ (also $d(e_i)$) is an extra start event for each $Ms(G, e)$. 

**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 |
| for each $b \in B$ do |
| if there is no $S \rightarrow x c(x) \in P$ such that $d(x) = b$ and there is no $y \in N$ such that $d(y) = b$ then |
| Select a k-sequence $e \in K$ such that $d(e) = b$ |
| $G' = G, M = M \cup \{Ms(G', e)\}, N = N \cup \{e\}$ |
| endfor |

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.

**Example 4.5 (Nonequivalence 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. Since there is no CES starting with a $p$ event in $G$, $Y = \emptyset$. Thus, $d(X) \neq d(Y)$ and $d(X)$ is an extra start event for each $e \in E$. Furthermore, each of these mutants models a different fault located at the mutation point; that is, $e_i$ (also $d(e_i)$) is an extra start event for each $Ms(G, e)$.

**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 $d(x) = d(e_i)$.
2. There is no previously selected mutation parameter $y$ such that $d(y) = d(e_i)$.

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, $e_i$ (also $d(e_i)$) is an extra start event for each $Ms(G, e)$.

**Example 4.4 (Usefulness and Determinism of a Mark Start Mutant).** Let $G$ be the 1-Reg in Fig. 1c. Since there is no CES starting with a $p$ event in $G$, $Y = \emptyset$. Thus, $d(X) \neq d(Y)$ and $d(X)$ is an extra start event for each $e \in E$. Furthermore, each of these mutants models a different fault located at the mutation point; that is, $e_i$ (also $d(e_i)$) is an extra start event for each $Ms(G, e)$.

**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 $e \in K$ such that $d(e) \in B$, $U = \{(a, e) \mid a \in \{a_1, \ldots, a_n\} \subseteq K\}$ and $V = \{(a, b) \mid b \in \{b_1, \ldots, b_n\} \subseteq K\}$, insert terminal (It) operator is defined as $\text{It}(G, \quad \varepsilon \quad V) = \{a, b \mid a \in U \quad \varepsilon \quad c(b) \to c(e) \mid (a, b) \in V\}$.

To generate mutants that contain a small number of changes, $|U| = 1$, that is, $U = \{(a, \varepsilon)\}$. Furthermore, since all null tests case are FCESS, $V = \emptyset$ and $c(e)$ is inserted for the usefulness of k-sequence $e$. Therefore, in the following, $G' = \text{It}(G, \quad \varepsilon \quad \emptyset)$, where $G'$ is selected as a mutation parameter if the following hold:

Example 4.7 (An Insert Terminal Mutant). Let $G$ be the 1-Reg in Fig. 1c. Fig. 5b shows $\text{It}(G, p_3, \{(p_2,p_3)\}$, $\emptyset$ where $p_3$ is a new contexted event. Note that, since $V = \emptyset$, $c(\varepsilon)$ 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

$$\text{T}_{\text{CES}}(G') = \text{T}_{\text{CES}}(G) \cup \{x \varepsilon \mid S \Rightarrow x c(a) \mid (a, e) \in E\}.$$

(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 $G' = \text{It}(G, p_3, \{(p_2,p_3)\}$, $\emptyset$ be the 1-Reg in Fig. 5b. The set of CESs is extended by event sequences that end with $p_3$, such as $x1 p_2 p_3$, $c1 x1 p_2 p_3$ and $c1 p1 x1 p_2 p_3$.

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 $d(x) \notin d(y)$ or $d(x') \notin d(y')$ where

- $X = \{x, e \mid S \Rightarrow x c(a) \mid (a, e) \in E\}$
- $Y = \{w \mid w \in T_{\text{CES}}(G) \text{ and } w \text{ contains } e' \mid e' \in K, e' \neq e \text{ and } d(x') = d(x)\}$

(Proof is included in Appendix.)

Example 4.9 (Equivalence of an Insert Terminal Mutant).

Let $G$ be the 1-Reg in Fig. 1c. $G' = \text{It}(G, p_3, \{(p_2,p_3)\}$, $\emptyset$ in Fig. 5b extends $T_{\text{CES}}(G)$ by $X$ that contains new sequences ending with $p_3$; these sequences actually end with $x1 p_2 p_3$. Although there are CESs in $G$ which contain a $p$ event, none of these sequences ends with an $x p p$ sequence. Thus, $d(x) \notin d(y)$ and $d(x)$, which means that there are additional contexted event sequences in $d(T_{\text{CES}}(G'))$. Thus, $d(T_{\text{CES}}(G')) \neq d(T_{\text{CES}}(G))$ and $G'$ is not equivalent to $G$.

The following gives 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 $G' = \text{It}(G, p_3, \{(p_2,p_3)\}$, $\emptyset$, the 1-Reg in Fig. 5b, is also useful. Furthermore, since $G$ is deterministic and no $p$ event follows $p_2$ 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).

Let $G$ be the 1-Reg in Fig. 1c, is useful, deterministic and no $p$ event follows $p_2$ in $G$. Therefore, its mutant $G' = \text{It}(G, p_3, \{(p_2,p_3)\}$, $\emptyset$, 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 $\text{It}(G, e, \{(a, e)\}, \emptyset)$, 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 $d(x) = d(e)$.
2. There is no previously selected mutation parameter $(y, \{(a, y)\}, \emptyset)$ such that $d(y) = d(e)$.

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, $c_1$ (also $d(e)$) is an extra event that follows k-sequence $a$ for each $\text{It}(G, e, \{(a, e)\}, \emptyset)$.

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 $e \in K$ do

$N = \emptyset$

for each $b \in B$ do

if there is no $c(a) \to x c(x) \in P$ such that $d(x) = b$ then

$b' = \text{a new contexted version of } b, e = a_1, \ldots, a_i$,

$G' = G, M = M \cup \{\text{It}(G', e, \{(a, e)\}, \emptyset)\}, N = N \cup \{e\}$

endfor

endfor

endfor

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 other insert terminal mutants.

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 $\text{It}(G, e, \{(a, e)\}, \emptyset)$ can be generated in $|P| + |B| + k = O(|P|)$ time by checking whether there are no $c(a) \to x c(x) \in P$ so that $d(x) = d(e)$ and previously selected mutation parameter $(y, \{(a, y)\}, \emptyset)$ so that $d(y) = d(e)$, preparing $e$ by copying $a_1, \ldots, a_i$ 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
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: $G = (E, B, K, C, P)$ – the k-Reg $(k \geq 1)$
Output: $X$ – a set of sequences covering faulty $(k+1)$-sequences

$X = \emptyset$, $Y = \emptyset$

for each $G' = Ms(G, c)$ selected as in Section 4.1 do
    $X = X \cup \{e\}$
endfor

for each $G' = It(G, c, \{(a, e)\}, \emptyset)$ selected as in Section 4.2 do
    $s = gen_seq(c, c) \rightarrow c(a)$ from $G'$
    $X = X \cup T_s(\{k\})$ //See Theorem 3.1
endfor
```

The time complexity of Algorithm 5 is given by $O(|B||P| + |K||B||P|)$ 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 $O(1)$ 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 (MK-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

The mixed k-Reg (M-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 (SSs) 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 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 bet-
ter 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 MK (Mutate and Kill)

The MK represents the mutate-and-kill-based (MK-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 (SUCs)

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.

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>K-REG MODELS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ShearBar</td>
</tr>
<tr>
<td>( k )</td>
<td>k-sequences / faulty k-sequences</td>
</tr>
<tr>
<td>1</td>
<td>314 / 103</td>
</tr>
<tr>
<td>2</td>
<td>395 / 32261</td>
</tr>
<tr>
<td>3</td>
<td>506 / 40574</td>
</tr>
<tr>
<td>4</td>
<td>626 / 51998</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 2</th>
<th>MUTANT NUMBERS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ShearBar</td>
</tr>
<tr>
<td>( k )</td>
<td>k-Reg Mutants</td>
</tr>
<tr>
<td>1</td>
<td>32364</td>
</tr>
<tr>
<td>2</td>
<td>40653</td>
</tr>
<tr>
<td>3</td>
<td>52077</td>
</tr>
<tr>
<td>Total</td>
<td>125094</td>
</tr>
</tbody>
</table>

---

\(^1\) http://www.claas.com

\(^2\) http://www.isik.de
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 Additional (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≥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 \( \sim 0.40\% \), \( \sim 2.97\% \) and \( \sim 3.62\% \) 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 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 \( \text{maxlen} \) 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, \( \text{maxlen} \) is limited to 60,63,67,70 for ShearBar and 20,30,40,50 for Specials and Additional. 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 \( \sim 25.07\% \). However, M-k-Reg almost always performs better than Random\((k+1)\) by revealing up to \( \sim 8.69\% \) 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 \( \sim 134.62\% \) and \( \sim 124.39\% \).

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 \( \sim 106.06\% \) 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, k-Reg and M-k-Reg are up to \( \sim 3.2 \) times and \( \sim 1.2 \) 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 \( \sim 99.99\% \). However, in some cases, test generation times for M-k-Reg are greater when compared to MK\((M-k)\), up to \( \sim 4.5 \) 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 \( \sim 70.65\% \) and \( \sim 33.62\% \) less test execution efforts than Random\((k+1)\).

In addition, Table 3 suggests that as \( k \) increases, test generation time increases, respectively, up to \( \sim 99.99\% \) to \( \sim 99.98\% \) less for k-Reg and M-k-Reg when compared to Random\((k+1)\). Furthermore, although it generally increases, respectively, up to \( \sim 99.96\% \) 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 \( \sim 67.09\% \) and \( \sim 17.26\% \) 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 Table 3, Table 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%, respectively. Furthermore, they are also always higher than MK(k) and MK(M-k), respectively, up to ~153.21% and ~144.12%.

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% faster decreasing trend than Random(k+1) and MK(k), and M-k-Reg shows, respectively, up to ~49.59% and ~155.57% 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.

**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 Table 4, Table 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% 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%. In addition, k-Reg and M-k-Reg are overall up to ~94.59% and ~180.00% 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 5% of the test execution for k-Reg and M-k-Reg and 5% for Random(k+1) and 5% for MK(k). 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 5% of the test execution and the second point resides around 50%.

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 5% lower than that of Random(k+1). This starts at some point after almost 80% 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 9.05% of the revealed faults for the last 5% of the execution. In addition, k-Reg and M-k-Reg achieve better FDRs than MK, except for the first ~1% to ~7% 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% more faults, respectively, than Random(k+1). Random(k+1) increases the number of revealed faults by detecting up to 7.87% of the revealed faults and executing up to 13.89% 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% 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(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% 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 hap-
pen, respectively, around ~60% to ~70% and ~80% to ~85% 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% 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%. 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% for k=1, from ~8.18% to ~15.19% for k=2, and from 0% to ~3.05% and from ~21.82% to ~27.13% 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% 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% of test execution for k=1. Also, M-k-Reg becomes up to ~29.66% less 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% for k=2, and from ~43% to ~88% 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%) of the test execution where it is similar to MK(M-k).

Additional. 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% 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 MK(k) and k-Reg, such sudden increases are more frequent, and they become more apparent as k gets larger. For M-k-Reg, the most significant increase happens around ~82% to ~87% of the test execution, and for k-Reg around ~55% to ~68%.

k-Reg and M-k-Reg are respectively up to ~90.80% and ~13.81% 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%. 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% (for k=1), ~12.69% (for k=2) and ~3.03% (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% 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% of the test execution; for k=2, M-k-Reg becomes up to ~21.74% less effective from ~48% to ~86% of the test execution; and, for k=3, it becomes up to ~34.78% less effective from ~37% to ~96% of the test execution. In addition, the FDR of M-k-Reg is less than MK(M-k) for the first ~28.27% (for k=1), ~6.80% (for k=2) and ~2.80% (for k=3) of the test execution, and it is greater in the res.

5.7 A Brief Summary of the Results

The new k-Reg approach detects on the average,
- for ShearBar, ~13% more faults per executed event;
- for Specials, ~143% more faults per executed event; and
- for Additionals, ~147% more faults per executed event,
when compared to the random testing approach, and,
- for ShearBar, ~72% more faults per executed event;
- for Specials, ~134% more faults per executed event; and
- for Additionals, ~105% 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% more faults per executed event;
- for Specials, ~29% more faults per executed event; and
- for Additionals, ~18% more faults per executed event,
when compared to the random testing approach, and,
- for ShearBar, ~76% more faults per executed event;
- for Specials, ~115% more faults per executed event; and
- for Additionals, ~103% 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%
(for ShearBar), ~2.97% (for Specials) and ~3.62% (for Add tionals) 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 \( \sim 1.32^{0.16k} \) 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 4 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, 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 (MBMT).

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, Black, and Majurski [8], and Black, Okun, and Yesha [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].

Bell, Budnik, and Wong [23] also use event-based models (ESGs) 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 Bell 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].
S \rightarrow \text{deposit ACC0} | \text{debit ACC0} \quad S \rightarrow \text{deposit ACC0} | \text{debit ACC0}
ACC0 \rightarrow \text{digit ACC1} \quad ACC0 \rightarrow \text{digit ACC1}
ACC1 \rightarrow \text{digit ACC2} \quad ACC1 \rightarrow \text{digit ACC2}
ACC3 \rightarrow \text{digit AMM0} \quad ACC3 \rightarrow \text{digit AMM0}
AMM0 \rightarrow S \ AMM1 \quad AMM0 \rightarrow S \ AMM1
AMM1 \rightarrow \text{digit AMM2} \quad AMM1 \rightarrow \text{digit AMM2} \ AMM2
AMM2 \rightarrow \text{digit AMM2} \quad AMM2 \rightarrow \text{digit AMM2}
AMM2 \rightarrow . \ AMM3 \quad AMM2 \rightarrow . \ AMM3
AMM3 \rightarrow \text{digit AMM4} \quad AMM3 \rightarrow \text{digit AMM4}
AMM4 \rightarrow \text{digit ACT} | \text{digit S} \quad AMM4 \rightarrow \text{digit ACT} | \text{digit S}
ACT \rightarrow \varepsilon \quad ACT \rightarrow \varepsilon

(a) Regular grammar. \quad (b) Context-free grammar.

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

### 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 Offutt et al. [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:

- 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% more faults per executed event than the random testing, and 72% to 134% more faults per executed event than the MK-based MBMT approach.
- **M-k-Reg** detected 16% to 29% more faults per executed event than the random testing, and 76% to 115% 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% 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.

### 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 who helped to adopt the mutate-and-kill-
based (MK-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.

APPENDIX

Appendix is available as supplemental material.

REFERENCES


