This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: F. Belli, M. Beyazıt, "A Formal Framework for Mutation Testing," The 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010), IEEE, Jun. 2010, pp. 121-130

A Formal Framework for Mutation Testing

Fevzi Belli, Mutlu Beyazıt
Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Paderborn, Germany
belli@adt.upb.de, beyazit@adt.upb.de

***Abstract*— Model-based approaches, especially based on directed graphs (DG), are becoming popular for mutation testing as they enable definition of simple, nevertheless powerful, mutation operators and effective coverage criteria. However, these models easily become intractable if the system under consideration is too complex or large. Moreover, existing DG-based algorithms for test generation and optimization are rare and rather in an initial stage. Finally, DG models fail to represent languages beyond type-3 (regular). This paper proposes a grammar-based mutation testing framework, together with effective mutation operators, coverage concepts and algorithms for test sequence generation. The objective is to establish a formal framework for model-based mutation testing which enables complementary or alternative use of regular grammars, depending on the preferences of the test engineer. A case study validates the approach and analyzes its characteristic issues.**

Keywords- (model-based) mutation testing; test coverage; test generation; directed graph; formal/regular grammar; manipulation operator

I. INTRODUCTION

Most of the model-based testing techniques operate on graphs, especially on directed graphs (DGs). This has been masterly expressed by one of the testing pioneers, Boris Beizer, as "Find a graph and cover it!" [1, 2]. The basic idea behind "graph coverage" entails generation of test cases and selection of a minimum number of them, called "test suite", in order to cost-effectively exercise a given set of structural or functional features of the software under test (SUT). A good test coverage increases user confidence in software artifacts, showing that the software is doing everything as it is supposed to do (*positive* testing, [3]).

For implementation-oriented, white-box testing, nodes of the DG to be covered usually represent the statements of SUT; arcs represent the sequences of those statements [4]. For specification-oriented, black-box testing, nodes of the DG may represent the behavioral events of SUT; arcs represent the sequences of those events [5].

When using a graph to model SUT, it is also proposed to cover not only the DG model given, but also its complement, showing that the software is *not* doing anything it is *not* supposed to do (*negative* testing, [5, 3]). For this purpose, the authors propose specific manipulation operators for the graph that models the SUT. Negative testing approach can be seen in relationship with mutation testing, which is originally considered as a white-box testing technique [6]. Recently, the mutation testing approach is extended to include black-box testing and model-based testing [7, 8].

A tough problem with complex SUTs is that modeling graphs rapidly become large and, thus, tedious to work with. Therefore the use of alternative or complementary structures can be beneficial. Since, with some proper precautions, it is possible to interpret the modeling DG as the transition diagram of a finite state automaton (FSA), and transform it into a regular expression (RE), and vice versa, using the well-known algorithms in the literature [9], it can be concluded that DGs can also be used to represent regular (type-3) languages. Thus, one can convert a DG to a corresponding grammar, more specifically to a regular grammar (RG), and work with this grammar instead of spacious graphs.

A DG is often quite compact, and, in practice, it is easier to work with a DG rather than a deterministic FSA, because only valid transitions of events are considered, and the states are not processed explicitly. From testing perspective, DG model-based mutation testing framework is quite simple and robust: Efficient operators can easily be defined to corrupt the sequencing of events, leading to fault models which are effective for systematic generation of mutants, enabling also definition of meaningful, sequence-based coverage criteria. Nevertheless, there are some limitations and problems related to the use of DGs, e.g., DGs can only be used to model regular systems. Moreover, test sequence generation algorithms based on DGs can be viewed as still being in a starting position, i.e., the existing few are relatively slow and memory-consuming. In this perspective, formal grammars can be utilized to overcome such and similar issues, e.g., to compactly model systems beyond regular, and introduce new algorithms.

In this paper, RGs are of primary interest, due to the underlying major and theoretical differences between RGs and higher level formal grammars, which effect their utilization in practice. Our research is motivated to introduce a novel, formally defined framework to be used in model-based mutation testing, and to overcome DG-related issues.

Present paper contributes to literature by establishing a complete RG model-based mutation testing framework. To our knowledge, this type of effective frameworks have not been proposed or discussed in detail for grammars.

This paper introduces new effective mutation operators for grammars which preserve the regularity, i.e., type-3 preserving. Based on common coverage notion, the concept of *k-sequence coverage criteria* ($k \geq 2$) for RGs is

introduced. Moreover, novel algorithms to generate test sequences related to this family of coverage criteria are proposed. Thus, at present stage of our research, the evaluation and the use of new grammar-related mutation concepts, test generation algorithms and test sequences are rendered possible in model-based mutation testing. Apart from this, DG-based and RG-based formal frameworks are examined and compared over a case study, which also validate the approach and analyzes its characteristic feature.

The rest of the paper is organized as follows: Next section summarizes relevant literature, before Section III defines DG-related notions used in this paper. Section IV introduces the RGs and the approach. Section V compares grammar-based and graph-based approaches over a case study, taking several factors into account. Finally, Section VI concludes the paper and sketches some ideas for future work.

II. Related Work

The current literature includes numerous well-known algorithms which can be used to convert a DG to a FSA and vice versa [10, 9, 11]. For both ways of conversion, the DG can be interpreted as a Moore-like machine [12] and the FSA can be interpreted a Mealy-like machine [13]. Furthermore, the case where the regular language includes the empty string should be handled with care. In case same symbol is used to label different nodes in the DG, an indexing mechanism [5, 7] is to be used to properly distinguish those nodes. This also establishes the DG-RE conversion, because the literature also outlines FSA-RE conversion.

DGs and related structures, which result from extension of DGs, are commonly used in areas like modeling, specification, validation and testing. Some examples are given as follows.

- Finite state machines, where [14] uses them to test correctness of control structures, [15] generates test cases for testing graphical user interfaces and [16] makes use of extended finite state machines for testing of interactive systems.
- Event sequence graphs (ESGs), where [5, 17] discuss test generation and minimization, [6] discusses generating mutation operators and generating mutants, and [3] shows integration with decision tables in testing process.
- Basic state charts, where [18] outlines test generation and minimization.
- View graphs, where [4] demonstrates their use in analysis and testing of programs.

In contrast to DGs, REs are generally not used standalone in practice although they still attract theoretical computer scientists. In general, REs are used as complementary or supportive tools to ease the representation of some elements. Researchers make use of REs in proposition of a software specification language including control and data structures [19], test case generation and selection [20, 21], fault detection and fault tolerance via Petri nets [22], and detection of faults in circuits [23]. Nevertheless, REs are still not extensively exploited; their expressive power and especially their rich algebraic properties offer for sure more. In such an attempt, [24], in a preliminary stage of the work, gives RE equivalents of DG manipulation operators to extend mutation testing concepts to REs.

On the other hand, grammars are another kind of constructs which are also used in the practice extensively. They have more expressive power than FSAs and REs, i.e., they can be used to represent not only regular, but also context-free and context-sensitive languages, etc. Their usage in practice includes compiler testing [25], test data and test case generation [26, 27], grammar testing [28] and mutation testing [29, 30].

The approach introduced in this paper differs from the ones mentioned above in that it

- defines efficient and effective (hierarchy preserving) RG-operators for (model-based) mutation testing,
- introduces novel coverage notions (for RGs),
- demonstrates their use in software testing, and
- compares the DG and RG-based frameworks.

III. Background

In this section, notions related to DGs are briefly introduced to structure the discussion in the following sections.

A. Basic Definitions

Usually, a directed graph is considered as a tuple (V, A) where V is a finite set of nodes and A is a finite set of directed arcs, which are unordered pairs of elements of V. However, while studying DGs from a formal linguistic perspective, e.g., while constructing RE of a given DG or DG of a given FSA, etc., some nodes in the graphs should be distinguished as *start* and *finish* nodes. This is done merely to enable semantic usefulness of the DG nodes and to represent the graph in a complete manner, in a given perspective, e.g., as introduced in ESGs (see [5], also Section V). Therefore, one can determine the start and the finish nodes (and so the nodes whose usefulness are required) according to the context in which DGs are used. Thus, unless noted otherwise, we shall comply with the following definition of DGs.

Definition 1: A *directed graph* (*DG*) is a tuple $D = (V, A, S, F)$ where

- V is a finite set of *nodes*,
- A is a finite set of *directed arcs* which are ordered pairs of elements of V, i.e., $A \subseteq V \times V = \{(u, v) \mid u, v \in V\}$,
- $S \subseteq V$ is a distinguished set of *start nodes*, and
- $F \subseteq V$ is a distinguished set of *finish nodes*.

Furthermore, a *path* is a sequence $x_1x_2\ldots x_k$ of nodes where each $(x_i, x_{i+1}) \in A$ for $i = 1, \ldots, k-1$. The *language* defined by DG D, denoted by $L(D)$, is the set of all paths which begin at a start node and ends at a finish node.

Definition 2: Given a DG (V, A, S, F) and a node $v \in V$, it is said that v is *useful* if and only if it occurs in a path from a start to a finish node.

Definition 3: Given a DG (V, A, S, F) and a node $v \in V$, *strictly preceding nodes* related to v are the nodes such that v

occurs in all the paths from these nodes to finish nodes. Furthermore, *strictly succeeding nodes* related to v are the nodes which only occur in the paths from v to finish nodes.

B. Directed Graph Manipulation Operators

For manipulation of a graph, two elementary operators can be applied: Insertion (i) and omission (o). As graphs consist of nodes and arcs, these operators can be specified as:

- *Node manipulation*
  - *Node insertion (i_n)*
  - *Node omission (o_n)*
- *Arc manipulation*
  - *Arc insertion (i_a)*
  - *Arc omission (o_a)*

Finally, the operators introduced above can be combined and/or carried out multiple times, which enable to transform any graph to any other one. Now, we introduce those notions more precisely.

Definition 4: DG *manipulation operators* transform a given DG to another DG, and are defined as follows:

- *Arc insertion* operator adds a new arc $a = (u, v)$ to the DG, where $u, v \in V$ and $(u, v) \notin A$. After the insertion, the new set of directed arcs is $A = A \cup a$.
- *Arc omission* operator deletes an existing arc $a = (u, v)$ from the DG where $u, v \in V$ and $(u, v) \in A$. After the omission, the set of directed arcs is updated as $A = A - a$. It is possible that the operation leaves some nodes with no ingoing and/or outgoing arcs.
- *Node insertion* operator adds a new node v to the DG together with possibly nonzero number of arcs, $a_1, \ldots, a_k$, connecting this node to the remaining nodes. After the insertion, the set of nodes is updated as $V = V \cup v$ and the new set of arcs is $A = A \cup a_1, \ldots, a_k$.
- *Node omission* operator deletes an existing node v from the DG together with the arcs, $a_1, \ldots, a_k$, ingoing to and outgoing from the deleted node. After the deletion, the set of nodes is updated as $V = V - v$ and the set of arcs is updated as $A = A - a_1, \ldots, a_k$.

C. Coverage

Three practical coverage criteria related to DGs are defined as below.

Definition 5: Given a DG $D = (V, A, S, F)$, a set of strings $B \subseteq L(D)$ is said to cover a node $v \in V$, if v occurs at least in one of the strings in B. If the set of string B covers all nodes in V, then it is said to achieve *node coverage*.

Definition 6: Given a DG $D = (V, A, S, F)$, a set of strings $B \subseteq L(D)$ is said to cover an edge $(u, v) \in A$, if the sequence uv occurs at least in one of the strings in B. If the set of string B covers all edges in A, then it is said to achieve *edge coverage*.

Definition 7: Given a DG $D = (V, A, S, F)$, a set of strings $B \subseteq L(D)$ is said to *cover* a path of length k, called *k-path*, $u_1u_2\ldots u_k$, if the sequence $u_1u_2\ldots u_k$ occurs at least in one of the strings in B. If the set of string B covers all k-paths, for some fixed $k > 0$, then it is said to achieve *k-path coverage*.

It is straightforward to note that k-path coverage is already a generalization of the edge coverage and node coverage, since *2*-path coverage is exactly the edge coverage and node coverage is equivalent to *1*-path coverage.

Discussion on the concept of coverage along with various coverage criteria in different contexts can be found in [31, 32, 28, 33, 34].

D. Notes on Validity and Usefulness

While working with manipulation operators, it may be beneficial to define validity criteria for the constructs or models under consideration, so that one can either perform only manipulation operations which do not invalidate the model or take additional measures to transform an invalidated model into a valid one.

In the previous section, our model is selected to be the set of DGs and a model is assumed to be valid as long as it is a DG. Therefore, the manipulation operators are not restricted, since all of them transform a given DG to another one preserving the validity of the model. However, in this discussion DGs are considered with no reference to any context, i.e., they are considered alone. In case one would like to construct DG model of a SUT, start and finish nodes should be determined with respect to the system semantics. In such cases, it also makes sense to define validity criteria which enforce usefulness (Definition 2) of all nodes in DG, in order to represent it in a complete manner. For the rest, there are two main approaches:

- Fix the set of start nodes and finish nodes and do not allow any sequence of manipulation operations which violate the usefulness of any node in DG.
- Perform a sequence of manipulation operations. If the resulting DG is invalid, then select new start and finish nodes to satisfy usefulness of all nodes and transform it into valid one.

Note that operators which may cause violation of DG validity are arc omission, node insertion or node omission, because only arc insertion operator preserves usefulness of (all the nodes in) DG and, so, the validity.

IV. Regular Grammars

This section outlines proper notions and extends the work on DGs to RGs. For this purpose, for RG, new effective manipulation operators are introduced, coverage concepts are discussed in detail and then extended, a distinguished form is defined and novel algorithms to generate strings achieving k-sequence coverage criteria are outlined.

A. Basic Definitions

Definition 8: A *formal grammar*, or just *grammar*, is a tuple (N, E, P, S) where

- N is a finite set of *nonterminal symbols*,
- E is a finite set of *terminal symbols*,
- P is a finite set of *production rules* of the form $Q \rightarrow R$ where $Q \in (N \cup E)^*N(N \cup E)^*$ and $R \in (N \cup E)^*$, and
- $S \in N$ is a distinguished nonterminal *start symbol*.

Furthermore, a *derivation step* is of the form $xQy \Rightarrow xRy$ with $x, y \in (N \cup E)^*$ and $Q \rightarrow R \in P$. A *derivation* is a sequence of derivation steps and denoted by $\Rightarrow^*$. The language

defined by grammar G, denoted by $L(G)$, is the set of strings $L(G) = \{w \in E^* \mid S \Rightarrow^* w\}$. A string w is called a *sentence* of G if $w \in L(G)$. Any string $R \in (N \cup E)^*$ such that $S \Rightarrow^* R$ is called a *sentential form* of G.

Definition 9: Given a grammar *(N, E, P, S)*, a terminal symbol r and a nonterminal symbol R. It is said that terminal symbol r is *useful* if it occurs in at least one string in *L(G)* and that nonterminal symbol R is *useful* if a rule of the form $(N \cup E)^* R (N \cup E)^* \rightarrow \ldots$ is used in a derivation $S \Rightarrow^* \ldots$ of at least one string in *L(G)*.

Definition 10: Given a grammar *(N, E, P, S)* and a terminal symbol r, *strictly preceding terminal symbols* related to r are the terminal symbols such that r occurs in all the strings in *L(G)* where these terminal symbols also occur and r occurs after them. Furthermore, *strictly succeeding terminal symbols* related to r are the terminal symbols which only occur in the strings and in at least one string in *L(G)* where r also occurs and they occur after r.

In present paper, we restrict the grammars to describe regular languages, and thus define a RG as follows.

Definition 11: Given a grammar $G = (N, E, P, S)$,

- G is said to be a *left regular grammar* if its production rules are in one of the following forms:
  $Q \rightarrow \varepsilon$, $Q \rightarrow s$ or $Q \rightarrow R\ r$, and
- G is said to be a *right regular grammar* if its production rules are in one of the following forms:
  $Q \rightarrow \varepsilon$, $Q \rightarrow s$ or $Q \rightarrow r\ R$.

where $r, s \in E$, $Q, R, S \in N$ and ε is the *empty string*. A *regular grammar (RG)* is a formal grammar which is either left regular or right regular.

B. Directed Graph - Regular Grammar Conversion

Since FSA, DGs and RGs equivalently describe regular languages, it is possible to construct RGs from DGs or FSA. Algorithms to convert a FSA to a RG are already given in the literature.

Similarly, a DG can be converted to a RG in $O(|V| + |S| + |F| + |A|)$ time by constructing an bijective mapping $nt(x) = R_x$ which maps a given terminal $x \in E$ to a nonterminal symbol R_x (Algorithm is skipped to save space). The obtained grammar has some distinguished properties.

Given a DG $D = (V, A, S, F)$, there is a grammar $G = (N, E, P, S)$ which satisfies the following properties:

- G is a right RG (thus unambiguous).
- $nt(x) = R_x$ is a bijection from E to $N \backslash \{S\}$.
- P does not have a production rule of the form $Q \rightarrow r$ where $r \neq \varepsilon$.
- P does not have a production rule of the form $S \rightarrow \varepsilon$.
- P does not have a production rule of the form $Q \rightarrow r\ S$, i.e., nonterminal S appears only on the left side of the production rule.

In addition, the following claims hold:

- Each node in V corresponds directly a terminal in E.
- Each arc in V corresponds to a production rule in P (including the pseudo arcs used to mark start and finish nodes).
- $|N| = |E| + 1 = |V| + 1$ and $|P| = |A| + |S| + |F|$.
- $\varepsilon \in E$ if and only if $\varepsilon \in V$.
- All terminal symbols in the grammar are useful if and only if all nodes in the DG are useful.

In general, to transform an arbitrary RG in such a way that it satisfies the above properties, one may need to make use of indexing mechanism, which often causes an increase in the number of terminal and nonterminal symbols in the grammar.

On the other hand, it is possible to convert a grammar satisfying the properties above to a DG in $O(|E| + |P|)$ time in a straightforward manner so that the above claims still holds.

It is evident that, in a similar manner, a DG can be converted to a left RG and vice versa.

C. Corresponding Manipulation Operators

In this section, algorithms to update RGs in order to reflect the changes incurred by application of manipulation operators are given. It is evident that operators introduced in this section are equivalent to operators defined in Section III.B in basis. Moreover, here, proper measures are taken to preserve the usefulness of all terminal symbols.

In the rest of the discussion, (unless otherwise noted) it is assumed that the RG under consideration possesses the properties outlined in Section IV.B.

1) Arc Insertion: Arc insertion operator (i_a) introduces a new arc to the given DG. Therefore, RG should be updated with a new production rule to include this arc. The resulting algorithm is very simple and given by Algorithm I.

ALGORITHM I. ARC INSERTION

Input: $G = (N, E, P, S)$ – right RG
(x, y) where $x, y \in E$ – arc to be inserted
Output: $G = (N, E, P, S)$ – updated grammar
$P = P \cup \{R_x \rightarrow y\ R_y\}$ //Add production rule

It is straightforward to note that Algorithm I runs in $O(1)$ time since arc insertion assumes that arc to be inserted is not already in the graph, therefore production rule to be inserted is not already in the grammar.

2) Arc Omission: Arc omission operator (o_a) removes an existing arc from the given DG. Therefore, removal of a production rule is necessary. In addition, some extra care is required because arc omission may violate the usefulness of some particular terminal symbols. Algorithm II outlines the effects of arc omission on the RG with measures to preserve usefulness of all the terminal symbols.

Algorithm II runs in $O(|P|)$ time because removal and testing membership of a production rule, and checking the existence of derivation $S \Rightarrow^* XyR_y$ $(X \in E^*)$ can all be performed in $O(|P|)$ steps.

After the removal of the rule $R_x \rightarrow y\ R_y$, terminal x and strictly preceding terminals related to terminal x may lose their usefulness, if terminal x is a strictly preceding terminal related to terminal y in original RG. In this case, although there exist derivations which start from S and produce them, these derivations do not terminate, because no derivation starting from S and consisting x results in a string of only

terminal symbols. This corresponds to the case when the strictly preceding nodes are reachable from at least one start node but there exist no path from any of them to a finish node in the DG. The usefulness of x can be preserved by addition of a new production rule $R_x \rightarrow \varepsilon$.

ALGORITHM II. ARC OMISSION

Input: $G = (N, E, P, S)$ – right RG
(x, y) where $x, y \in E$ – arc to be removed
Output: $G = (N, E, P, S)$ – updated grammar

$P = P \setminus \{ R_x \rightarrow y R_y\}$ //Remove production rule
if $R_x \rightarrow Q \notin P$ for all right hand sides $Q \neq x R_x$ **then**
$P = P \cup \{R_x \rightarrow \varepsilon\}$ //Preserve usefulness of x
endif
if there exist no $S \Rightarrow^* XyR_y$ $(X \in E^*)$ **then**
$P = P \cup \{S \rightarrow y R_y\}$ //Preserve usefulness of y
endif

On the other hand, if terminal y is a strictly succeeding terminal related to terminal x in original RG, the usefulness of y becomes violated. More precisely, terminal y and the strictly succeeding terminals related to terminal y do not occur in any string of the new language, due to the fact that, although there exist derivations starting from R_y which may result in terminal strings containing them, no derivation starting with S and resulting in a string of only terminal symbols produces them. This stems from the fact that the production rules of the form $R_y \rightarrow \ldots$ may no longer be used in these derivations. In this case, to preserve the usefulness of y (and R_y) a new production rule of the form $S \rightarrow y R_y$ is added.

3) Node Insertion: Node insertion operator (i_n) is based on arc insertion because it entails adding a new node to the DG and connecting this node to the rest of the graph by inserting arcs ingoing to and outgoing from this new node. Naturally, insertion of a new node requires adding a new nonterminal symbol and a new terminal symbols (for the node), and new productions rules (for the arcs). Furthermore, additional production rules may be required to establish the usefulness of the new node. The steps to update the RG with the changes stemming from a node insertion are shown in Algorithm III.

Running time complexity of Algorithm III is given by $O(s+t)$ where s is the number of outgoing arcs and t is the number of ingoing arcs to be inserted. Note that a looping arc is considered to be an outgoing arc and, therefore, $(s+t) \leq 2|E|+1 = 2|V|+1$.

To sum up, Algorithm III adds new terminal v to E and corresponding nonterminal R_v to N. Later, for each arc to be inserted, a new production rule is added by the arc insertion algorithm (see Algorithm II). As the final steps, the usefulness of the new terminal is guaranteed: The production rule $R_v \rightarrow \varepsilon$ is added to P if derivations starting from R_v does not terminate (there is no outgoing arc from v in the DG, therefore v is marked as a finish node), and the production rule $S \rightarrow v R_v$ is added to P if derivations starting from does produce v or R_v (there is no ingoing arc to v in the DG, therefore v is marked as a start node).

ALGORITHM III. NODE INSERTION

Input: $G = (N, E, P, S)$ – right RG
v – node to be inserted
(v, x_j), where $x_j \in E \cup \{v\}$, $j = 1, \ldots, s$ – outgoing arcs
(y_k, v), where $y_k \in E$, $k = 1, \ldots, t$ – ingoing arcs
Output: $G = (N, E, P, S)$ – updated grammar

$E = E \cup \{v\}$ //Add new terminal symbol v
$N = N \cup \{R_v\}$ //Add new nonterminal symbol R_v
for each (v, x_j) **do**
perform insertion of (v, x_j) on G //See Algorithm I
endfor
for each (y_k, v) **do**
perform insertion of (y_k, v) on G //See Algorithm I
endfor
if $s < 1$ **or** $(s=1$ **and** $x_1=v)$ **then**
$P = P \cup \{R_v \rightarrow \varepsilon\}$ //Establish usefulness of v
endif
if $t < 1$ **then**
$P = P \cup \{S \rightarrow v R_v\}$ //Establish usefulness of v
endif

4) Node Omission: Node omission operator (o_n) is based on arc omission because in order to remove a node all the arc ingoing to and outgoing from this node need to be removed. Thus, the proper steps would be: First perform arc omission operation to remove the production rules, and then remove the isolated terminal symbol from the grammar together with the corresponding nonterminal symbols and the remaining production rules. Algorithm outlining these steps is given in Algorithm IV.

ALGORITHM IV. NODE OMISSION

Input: $G = (N, E, P, S)$ – right RG
v – node to be omitted
(v, x_j), where $x_j \in E \cup \{v\}$, $j = 1, \ldots, s$ – outgoing arcs
(y_k, v), where $y_k \in E$, $k = 1, \ldots, t$ – ingoing arcs
Output: $G = (N, E, P, S)$ – updated grammar

for each (v, x_j) **do**
perform omission of (v, x_j) on G //See Algorithm II
endfor
for each (y_k, v) **do**
perform omission of (y_k, v) on G //See Algorithm II
endfor
$E = E \setminus \{v\}$ //Remove terminal symbol v
$N = N \setminus \{R_v\}$ //Remove nonterminal symbol R_v
$P = P \setminus \{S \rightarrow v R_v, R_v \rightarrow \varepsilon\}$ //Remove production rules for v and R_v

Algorithm IV terminates in $O((s+t)\ |P| + |E| + |N|)$ number of steps where s is the number of arcs outgoing from v and t is the number of arc ingoing to v to be omitted. Note that a looping arc is considered to be an outgoing arc and, therefore, $(s+t) \leq 2|E|-1 = 2|V|-1$.

Algorithm IV first omits all the arcs related to node v, i.e., production rules related to terminal v are removed, by subsequent applications of arc omission algorithm. After this step, P contains only two rules related to v which preserve the usefulness of terminal v. These rules are $S \rightarrow v R_v$ and $R_v \rightarrow \varepsilon$. The remaining steps required to complete the node omission are to remove terminal v from E, to remove

nonterminal R_v from N, and to remove $S \rightarrow v\ R_v$ and $R_v \rightarrow \varepsilon$ from P.

D. Coverage and String Generation

As already mentioned, an important concept which is related to the use of structures as to graphs and grammars in - software testing and grammar testing is the concept of coverage. This section outlines some of the coverage criteria for right RGs in our consideration, which are also implied by the DGs.

1) Simple Coverage Criteria: There are two immediate coverage criteria associated with the grammars: terminal symbol coverage and production rule coverage, which can be defined as follows.

Definition 12: Given a grammar $G = (N, E, P, S)$, a set of strings $A \subseteq L(G)$ is said to cover a terminal symbol $e \in E$, if e occurs at least in one of the strings in A. If the set of string A covers all terminal symbols in E, then it is said to achieve *terminal symbol coverage*.

Definition 13: Given a grammar $G = (N, E, P, S)$, a set of strings $A \subseteq L(G)$ is said to cover a production rule $p \in P$, if p is used at least once in a derivation of a string in A. If the set of strings A covers all production rules in P, then it is said to achieve *production rule coverage*.

Note that, for the grammars and the graphs in our consideration,

- terminal symbol coverage for RGs is equivalent to node coverage for DGs, and
- production rule coverage is equivalent to edge coverage.

Furthermore, achieving production rule coverage is a sufficient condition for achieving terminal symbol coverage.

2) k-sequence Coverage: In order to continue our discussion by making an analogy to k-path coverage in DGs, k-sequence coverage and tail-bijective k-sequence right RGs are introduced and defined as follows.

Definition 14: Given a grammar $G = (N, E, P, S)$, a set of strings $A \subseteq L(G)$ is said to cover a sequence of length k ($k \geq 1$), $u_1 \ldots u_k$ such that $S \Rightarrow^* Xu_1 \ldots u_k Y$ for $X, Y \in (N \cup E)^*$, if $S \Rightarrow^* Xu_1 \ldots u_k Y$ is used at least once in derivation of a string in A. If the set of strings A covers all such k-sequences, then it is said to achieve *k-sequence coverage*.

Note that k-sequence coverage for the right RGs corresponds to k-path coverage for the DGs ($k \geq 1$).

Definition 15: Given a grammar $G = (N, E, P, S)$, G is said to be a *tail-bijective* (*t-b*) *k-sequence* (*k-seq*) *right regular grammar* ($k \geq 1$) if its production rules are in one of the following forms:

$$Q \rightarrow \varepsilon \quad or \quad Q \rightarrow r\ R_r$$

where

- $r = r(1) \ldots r(k) \in E$, $Q \in N$ and $R_r \in N \backslash \{S\}$,
- the function $nt(x) = R_x$ defines a bijection from E to $N \backslash \{S\}$,
- if $(Q \rightarrow r\ R_r) = (R_q \rightarrow r\ R_r)$ then $q(2) \ldots q(k) = r(1) \ldots r(k-1)$ where $R_q \in N \backslash \{S\}$, and
- ε is the empty string.

As Definition 15 implies, E is no longer a set of terminal symbols but a set of "terminal" strings of length k. Furthermore, note that right RGs which satisfy the properties defined in Section IV.B are t-b *1*-seq right RGs, because for $(Q \rightarrow r\ R_r) = (R_q \rightarrow r\ R_r)$, $q(2) \ldots q(k) = r(1) \ldots r(k-1) = \varepsilon$ if and only if $k \leq 1$. In addition, given a t-b k-seq right RG G, there exist $w \in L(G)$ such that $|w| = 2k$, if and only if there are at least two different rules such that $R_z \rightarrow x\ R_x$ and $R_x \rightarrow y\ R_y$.

Algorithm V shows the steps to transform a t-b k-seq right RG to a t-b $k+1$-seq right RG.

ALGORITHM V. TRANSFORMATION OF A TAIL-BIJECTIVE *K*-SEQUENCE RIGHT REGULAR GRAMMAR

Input: $G_k = (N, E, P, S)$ – t-b k-seq right RG
Output: $G_{k+1} = (N_{k+1}, E_{k+1}, P_{k+1}, S)$ – t-b $k+1$-seq right RG

$E_{k+1} = \varnothing$, $N_{k+1} = \{S\}$, $P_{k+1} = \varnothing$
for each $A \in P$ **do**
  if $A = R_z \rightarrow x\ R_x$ where z = $z(1) \ldots z(k)$ and $x = x(1) \ldots x(k)$ **then**
    $E_{k+1} = E_{k+1} \cup \{zx(k)\}$
    $N_{k+1} = N_{k+1} \cup \{R_{zx(k)}\}$
  endif
  for each $B \in P$ **do**
    if B = $R_x \rightarrow y\ R_y$ where $y = y(1) \ldots y(k)$ **then**
      if $A = R_z \rightarrow x\ R_x$ **then**
        $P_{k+1} = P_{k+1} \cup \{R_{zx(k)} \rightarrow xy(k)\ R_{xy(k)}\}$
      else if $A = S \rightarrow x\ R_x$ **then**
        $P_{k+1} = P_{k+1} \cup \{S \rightarrow xy(k)\ R_{xy(k)}\}$
      else if $A = R_y \rightarrow \varepsilon$ **then**
        $P_{k+1} = P_{k+1} \cup \{R_{xy(k)} \rightarrow \varepsilon\}$
      endif
    endif
  endfor
endfor

In Algorithm V, all set union operations can be performed in $O(1)$ time as append operations, due to the fact that during each respective union operation, a different element is added to the corresponding set. Thus, worst case time complexity of the algorithm is given by $O(k|P|^2)$. However, it is possible to achieve $O(|P|^2)$ running time if the corruption of the input grammar terminal strings is allowed.

As side remarks, it is possible to restrict t-b k-seq right RGs so that the languages defined by them do not contain any string of length k or to extend t-b k-seq right RGs so that the languages defined by them also includes all strings of length $< k$ which can not be included as a substring but are in the language defined by corresponding t-b *1*-seq grammar. However, these restrictions or extensions require proper modifications in the transformation algorithm.

Another important property of the strings in the language described by a t-b k-seq right RG can be outlined as follows: Let $G = (N, E, P, S)$ be t-b k-seq right RG and $s \in L(G)$ be a string, then

- $|s| = xk$ for some integer $x \geq 1$, and
- $s(i)(2) \ldots s(i)(k) = s(i+1)(1) \ldots s(i+1)(k-1)$ where $s = s(1) \ldots s(x)$ and $s(i)$ are disjoint and consequent substrings of s of length k.

There exist an $O(k + k\ (|s| - k)) = O(|s|\ k)$ time algorithm $T(s)$ which transforms a given string so that it satisfies the above properties (Algorithm is skipped to save space).

Now, let $G = (N, E, P, S)$ be t-b *1*-seq right RG and A be a set of strings. Furthermore, let $G_k = (N_k, E_k, P_k, S)$ be the t-b k-seq right RG obtained by repeated *k-1* applications of Algorithm V on G and $T(A)$ be the set of strings which is obtained by the application of $T(s)$ on the strings in A:

- $k+1$-sequence coverage for grammar G is achievable if and only if $L(G_k)$ contains at least one string of length $2k$, and
- A achieves $k+1$-sequence coverage for grammar G if and only if $T(A)$ achieves production rule coverage for grammar G_k.

3) String Generation: Given the condition to establish k-sequence coverage for a given right RG, i.e., t-b *1*-seq right RG, an algorithm to produce a set of strings which achieves k-sequence coverage can easily be constructed by making use of well known sentence generation algorithms [35, 36, 37] which achieve production rule coverage.

Before, outlining the steps to produce a set of strings which achieves k-sequence coverage ($k \geq 2$), Algorithm VI, which basically functions as an inverse transformation of $T(s)$, is given below.

ALGORITHM VI. $T^{-1}(s)$ - INVERSE TRANSFORMATION OF A STRING

Input: s – a string of length xk where integer $x \geq 1$
Output: s' – a strings of length $k+ x - 1$

$l = k + length(s)/k - 1$
s' is a string of length l
$s'(1)...s'(k) = s(1)...s(k)$
for $i=k+1$ **to** l
  $d = (i-k)*k$
  $s'(i) = s(d+k)$
endfor

Worst case running time of Algorithm VI, $T^{-1}(s)$, is $O(k + |s|/k)$ where $|s|$ is an upper bound on the input string length.

Algorithm VII makes use of these inverse transformations in order to generate a set of strings which achieves k-sequence coverage for the given *1*-sequence RG.

ALGORITHM VII. GENERATING A STRING SET ACHIEVING K-SEQUENCE COVERAGE

Input: $G = (N, E, P, S)$ – t-b *1*-seq right RG
  k – an integer ≥ 2
Output: A – a strings set which achieves k-sequence coverage for G

$A = \varnothing$
$G_{k-1} = G$
for $i=2$ **to** $k-1$
  G_{k-1} = transform G_{k-1} //See Algorithm V
endfor
A' = generate string set achieving production rule coverage for G_{k-1}
for each $a' \in A'$ such that $|a'| \geq 2k$ **do**
  $A = A \cup T^{-1}(a')$ //See Algorithm VI for $T^{-1}(.)$
endfor

It is clear that, although several efficient sentence generation algorithms exist for grammars, performance of Algorithm VII is relatively poor, especially for large k.

V. CASE STUDY

In this section, to compare the properties of the strings generated by DGs and RGs, a case study on a real-life application is performed (meanwhile also the applicability of the proposed approach is demonstrated). In this case study, we choose to compare generated test sequences, because algorithms of different nature are used in each framework and outputs are likely to have different properties. In this perspective, obtained results are supposed to give some insight on the respective use of discussed frameworks. Furthermore, the way comparisons are made may be used to establish some quality criteria for generated sequences or, in some sense, for mutants with respect to the used framework and nature of the system.

In the case study, test sequences generated using both DG and RG models to satisfy k-sequence coverage ($k = 2, 3, 4$). These sequences are compared in terms of minimization, tractability and higher sequence coverage capability.

Minimization is important to keep the costs of the testing process to a minimum. However, the complete minimization is not always desired. In engineering practice, although there is not always a clear formalization, it is often necessary to introduce some redundancy to increase the confidence level and gain performance.

Tractability gives some idea on how it becomes easy to follow or track test sequences during execution. If a periodic validation is required after some steps in test case execution, tractability of a sequence becomes more important especially for the test oracle.

Higher sequence coverage capability outlines how well test sequences cover a set of sequences which are not really expected to be covered. Thus, it also yields some insight on error detection capability and power of test sequences.

A. ISELTA System

ISELTA is an online reservation system for hotel providers and agents. It is a cooperative product of the work between a mid-size travel agency (ISIK Touristik Ltd.) and University of Paderborn. For our case study, we will consider a relatively small part of ISELTA called "Specials Module". Through this module, one is able to add special prices to the specified number of rooms of certain type for the determined period of time in the given hotel. Consequently, one can edit existing specials and also remove them.

Figure 1 demonstrates a simplified DG, interpreted as an event sequence graph (ESG) model of the considered specials module. Nodes of an ESG are considered as events, and start and finish nodes are unique pseudo nodes [5]. Edges connect nodes and represent the subsequences of events.

The right RG of SUT in Figure 1 has 11 terminal symbols, 12 nonterminal symbols and 48 production rules some of which are as follows (nt: $E \rightarrow N \setminus \{S\}$ is a bijection):

- $S \rightarrow$ EnterSpecials nt(EnterSpecials)
- nt(EnterSpecials) $\rightarrow$ Data_1 nt(Data_1)
- nt(ExitSpecials) $\rightarrow \varepsilon$

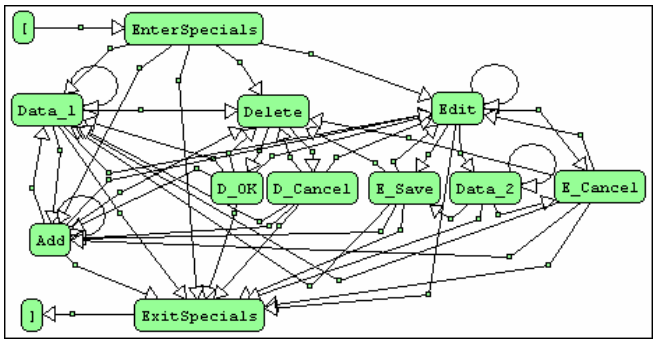

Figure 1. DG Model for Specials Module.

B. Comparison of Generated Test Sequences

The sets of test sequences are generated from 5 different DGs and RGs:

- O: Original model
- M1: Mutant model - insertion of (Edit, Delete) arc
- M2: Mutant model - insertion of (Delete, Add) arc
- M3: Mutant model - insertion of Delete_2 node with (Edit, Delete_2) and (Delete_2, Add) arcs
- M4: Mutant model - insertion of (Edit, Delete) and (Delete, Add) arcs

The mutants are not equivalent and their orders are kept small due to the competent programmer hypothesis [38] and the coupling effect [6], which mutation testing relies on. In this sense, arc or node removed mutants are discarded (assuming a proper fault model), because, in our case, sole applications of arc or node omission tend produce a correct sub model, leading to highly redundant test sets.

In addition, in order to generate test sequences from DGs, *Chinese Postman Problem* is solved [39, 17], and for RGs, *Purdom's algorithm* is employed [36, 40]. Also, although it is not our aim to make a performance comparison of the algorithms, we could not help but notice that (even a non- or sub-optimized implementation of) RG-based test sequence generation algorithm is much faster than that of DG especially for greater k and larger models. Furthermore, we noted that both DG and RG algorithms may produce slightly different outputs when order of input elements change.

1) Length Analysis: To perform the length analysis on the given models using DGs and RG structures, generated test sequences achieving k-sequence coverage ($k = 2, 3, 4$) are processed to collect the statistics given in Table I.

2) Sequence Coverage Anaysis: Algorithms to generate test sequences from both DGs and RGs are designed to achieve k-sequence (k-path) coverage. Nevertheless, it might be useful to investigate how well they inherently achieve k'-sequence coverage (covering symbol k'-tuples) for $k' > k$ in order to derive more information on the quality with respect to the used frameworks. Table II demonstrates such coverage data for $k = 2, 3, 4$ and $k' = 3, 4, 5, 6$ (for $k' > k$).

C. Results

Table I shows that number of sequences generated from RG mutants and their total length are greater. One can deduce that RG sequences contain roughly 25% to 50% redundancy. This directly implies that minimization level of RG algorithm is not as good as that of DG algorithm which performs complete minimization. On the other hand, test sequences generated from DG mutants are, on the average longer, and sequence lengths of RG mutants are closer to each other. This leads to the fact that RG sequences have better tractability properties. These arguments hold for sequences with fixed $k = 2, 3, 4$.

In addition, when the rate of change on the number, the total length and the average length of test sequences are considered for increasing k, in almost all cases, the respective rate in RG framework seems to be quite close to that of DG framework.

Table II outlines some more interesting results. Independent of the framework, a test set achieving k-sequence coverage covers roughly 32% to 36% of $k+1$-sequences, 7% to 10% of $k+2$-sequences, %1 to %2 of $k+3$-sequences and less than 0.5% of $k+4$-sequences, and these intervals stay the same regardless of k values. In addition, for low k (*2* or *3*), RG sequences cover a greater number of $k+1$-sequences when compared to DGs. However, this situation is reversed for higher k values or while trying to cover longer sequences of length $k+2$ or above.

TABLE I. DATA ON LENGTHS OF GENERATED TEST SEQUENCES

Model	Structure	String Number			Total Length			Min. Length			Max. Length			Average Length			Standard Deviation		
		k=2	*k=3*	*k=4*	*k=2*	*k=3*	*k=4*	*k=2*	*k=3*	*k=4*	*k=2*	*k=3*	*k=4*	*k=2*	*k=3*	*k=4*	*k=2*	*k=3*	*k=4*
O	DG	*9*	*31*	*115*	*79*	*322*	*1328*	*2*	*2*	*3*	*20*	*29*	*47*	*8.78*	*10.39*	*11.55*	*2.11*	*1.18*	*0.74*
	RG	*18*	*60*	*254*	*104*	*419*	*1912*	*2*	*2*	*3*	*15*	*26*	*22*	*5.78*	*6.98*	*7.53*	*0.84*	*0.49*	*0.16*
M1	DG	*9*	*31*	*117*	*80*	*338*	*1425*	*2*	*2*	*3*	*23*	*31*	*68*	*8.89*	*10.90*	*12.18*	*2.41*	*1.32*	*0.91*
	RG	*20*	*62*	*278*	*112*	*446*	*2114*	*2*	*2*	*3*	*15*	*20*	*32*	*5.60*	*7.19*	*7.60*	*0.65*	*0.50*	*0.19*
M2	DG	*9*	*32*	*125*	*81*	*348*	*1516*	*2*	*2*	*3*	*20*	*30*	*59*	*9.00*	*10.88*	*12.13*	*2.07*	*1.25*	*0.79*
	RG	*21*	*67*	*294*	*115*	*460*	*2194*	*2*	*2*	*3*	*19*	*17*	*26*	*5.48*	*6.87*	*7.46*	*0.74*	*0.40*	*0.15*
M3	DG	*9*	*32*	*119*	*82*	*358*	*1495*	*2*	*2*	*3*	*23*	*34*	*54*	*9.11*	*11.19*	*12.56*	*2.32*	*1.34*	*0.85*
	RG	*18*	*66*	*281*	*104*	*468*	*2150*	*2*	*2*	*3*	*16*	*35*	*30*	*5.78*	*7.09*	*7.65*	*0.77*	*0.56*	*0.18*
M4	DG	*9*	*32*	*128*	*82*	*366*	*1647*	*2*	*2*	*3*	*23*	*40*	*47*	*9.11*	*11.44*	*12.87*	*2.49*	*1.60*	*0.84*
	RG	*20*	*64*	*327*	*113*	*468*	*2461*	*2*	*2*	*3*	*17*	*22*	*24*	*5.65*	*7.31*	*7.53*	*0.69*	*0.46*	*0.15*

TABLE II. *K*-SEQUENCE COVERAGE OF GENERATED SETS OF TEST SEQUENCES

Model	Structure	*3*-sequence Coverage			*4*-sequence Coverage			*5*-sequence Coverage			*6*-sequence Coverage		
		k=2	*k=3*[a]	*k=4*[a]	*k=2*	*k=3*	*k=4*[a]	*k=2*	*k=3*	*k=4*	*k=2*	*k=3*	*k=4*
O	DG	*0.3518*	-	-	*0.0898*	*0.3441*	-	*0.0216*	*0.0939*	*0.3587*	*0.0052*	*0.0231*	*0.0979*
	RG	*0.3642*	-	-	*0.0847*	*0.3508*	-	*0.0174*	*0.0873*	*0.3540*	*0.0036*	*0.0177*	*0.0860*
M1	DG	*0.3430*	-	-	*0.0839*	*0.3401*	-	*0.0196*	*0.0907*	*0.3457*	*0.0046*	*0.0216*	*0.0931*
	RG	*0.3605*	-	-	*0.0823*	*0.3478*	-	*0.0146*	*0.0853*	*0.3478*	*0.0026*	*0.0173*	*0.0829*
M2	DG	*0.3391*	-	-	*0.830*	*0.3394*	-	*0.0192*	*0.0884*	*0.3492*	*0.0044*	*0.0211*	*0.0924*
	RG	*0.3621*	-	-	*0.0799*	*0.3363*	-	*0.0136*	*0.0792*	*0.3428*	*0.0023*	*0.0152*	*0.0789*
M3	DG	*0.3352*	-	-	*0.0863*	*0.3482*	-	*0.0205*	*0.0964*	*0.3606*	*0.0049*	*0.0236*	*0.0992*
	RG	*0.3470*	-	-	*0.0801*	*0.3574*	-	*0.0151*	*0.0885*	*0.3572*	*0.0029*	*0.0176*	*0.0873*
M4	DG	*0.3297*	-	-	*0.0766*	*0.3228*	-	*0.0170*	*0.0831*	*0.3334*	*0.0039*	*0.0192*	*0.0868*
	RG	*0.3405*	-	-	*0.0739*	*0.3311*	-	*0.0125*	*0.0755*	*0.3345*	*0.0020*	*0.0141*	*0.0748*

a. Values in this column are often (very close to) 1.0 and it is quite easy to include additional test sequences to achieve full coverage without significantly changing the results.

Finally, for test sequences generated from both DG and RG structures, *k'*-sequence coverage ratio rapidly decreases as *k'* increases for any fixed *k*.

D. Lessons Learned

To sum up, the use of DGs seems more promising. However, related algorithms suffer from (yet) low performance. While working with coverage of longer sequences, performance difference becomes more apparent. Therefore, if one requires faster generation of test sequences and redundancy is tolerable, RG-based framework poses a better choice. On the other hand, if absolute minimization is desired, the use DGs is inevitable.

Both structures have their use in practice. As an example, one may prefer to use DG if initializing the system for a test sequence execution is costly or requires relatively greater effort. On the other hand, it may be more suitable to use RGs, if one prefers more tractable test sequences and increased confidence level in testing process. Furthermore, it is possible to reduce the number of test sequences by prioritization and perform selective execution, while keeping the redundancy at a reasonable level.

E. Threats to Validity

Our experiments aim to give some insight on the characteristics of DG-based and RG-based frameworks comparatively, instead of drawing absolute conclusions.

There are several reasons for this: First, there are numerous real-life applications which possess different properties. Thus, results of an application may not be valid for one another. Second, it is assumed that SUT can be modeled using regular languages. Although this does not always hold, approximate regular models can be used to increase efficiency for many applications. Next, number and order of the mutants are limited for practical reasons. It would be interesting to see how the frameworks fare with increased number of higher order mutants from theoretical and practical perspectives. Also, in case study, only sequence-based coverage criteria are considered. It is still possible to define and achieve different coverage criteria by to making use of these structures.

VI. CONCLUSION AND FUTURE WORK

The purpose of this paper is not to (re)define and demonstrate the methodology and the mutation analysis concepts (like dead or live mutants etc.) for model-based mutation testing process. Current literature, e.g. [7, 17], already outlines the elements required for models, like DGs, to be utilized in such a testing process. Consequently, RGs together with notions introduced here can also be used in the same fashion. This is, however, not the primary objective of this paper.

The primary objective of this paper is to take an important and necessary step to extend model-based mutation testing concepts to formal grammars by outlining them in regular domain. For this purpose, throughout the paper, RGs with special properties, hierarchy preserving RG manipulation operators, testing-related coverage concepts for RGs, and algorithms to generate sets of strings to achieve a new family of coverage criteria are considered. To our knowledge, no other work introduced them. i.e., the proposed notions are entirely novel. Furthermore, the introduced mutation operators are effective in the sense that they preserve regularity and efficiently produce mutants resulting from different, systematic sequencing of symbols or system events, especially under properly defined usefulness criteria.

As the case study shows, (in regular domain) for test sequence generation, although DG-based framework produces minimized results, RG sequences have better tractability properties. In addition, the redundancy introduced by RG framework can be used to increase the confidence. In terms of coverage capability, RG sequences have better coverage for low *k* and adjacent *k'*(=*k+1*) values. Also, RG-based generation algorithms are much faster in general.

Considering the languages beyond regular domain, the use of grammars is inevitably more appropriate. Therefore, further research along this study includes generalization and extension of the established grammar-based framework to include context-free (and other formal) languages. However, one needs to be cautious while performing this kind of generalization, due to the fact that regular and more general languages are quite different from the theoretical perspective. For example, several decidability properties which hold for

regular languages are not valid even for context-free languages, and this may pose serious problems while trying to define hierarchy preserving mutation operators for more general languages or grammars. In addition, we also propose defining different coverage criteria and related algorithms, investigating the effects of input (re)ordering on the outputs of test sequence generation algorithms together with detailed performance analyses and developing a full-fledged testing tool employing the introduced concepts as potential and attractive future studies.

REFERENCES

[1] B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, 1990.

[2] R. V. Binder, Testing Object-Oriented Systems, Addison-Wesley, 2000.

[3] F. Belli, M. Linschulte, "On 'Negative' Tests of Web Applications," Annals of Mathematics, Computing & Teleinformatics, vol. 1, no. 5, 2008, pp. 44-56.

[4] S. Gossens, F. Belli, S. Beydeda, M. D. Cin, "View Graphs for Analysis and Testing of Programs at Different Abstraction Levels," Proc. of the 9th International Symposium on High-Assurance Systems Engineering (HASE 2005), IEEE CS Press, Oct. 2005, pp. 121-13.

[5] F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces," Proc. of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001), IEEE CS Press, Nov. 2001, pp. 34-43.

[6] R. A. DeMillo, R. J. Lipton, F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," IEEE Computer, vol. 11, no. 4, Apr. 1978, pp. 34–41.

[7] F. Belli, C. J. Budnik, E. Wong, "Basic Operations for Generating Behavioral Mutants," Proc. of the 2nd Workshop on Mutation Analysis in conjunction with ISSRE'06, IEEE CS Press, Nov. 2006, pp. 9.

[8] R. Wang, N. Huang, "Requirement Model-Based Mutation Testing for Web Service," 4th International Conference on Next Generation Web Services Practices (NWESP 2008) , Oct. 2008, pp.71-76.

[9] A. Salomaa, I. N. Sneddon, Theory of Automata, 1969.

[10] A. Gill, Introduction to the Theory of Finite-State Machines, 1962.

[11] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages and Computation (2nd Edition), 2000.

[12] E. F. Moore, "Gedanken Experiments on Sequential Machines," In Automata Studies, Ann. of Math. Studies No. 34, Princeton U. Press, 1956, pp. 129-153.

[13] G. H. Mealy, "A Method for Synthesizing Sequential Circuits," Bell System, Tech. J 34, Sep. 1955, pp.1045-1079.

[14] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," IEEE Transactions on Software Engineering, vol. SE-4, no. 3, May 1978, pp. 178-187.

[15] L. White, H. Almezen, "Generating Test Cases for GUI Responsibilities using Complete Interaction Sequences," Proc. of the 11th International Symposium of Software Reliability Engineering (ISSRE 2000), 2000, pp. 110-121.

[16] M. Fantino, M. Jino, "Applying Extended Finite State Machines in Software Testing of Interactive Systems," Design, Specification, and Verification in Interactive Systems, Lecture Notes in Computer Science, vol. 2844, 2003, pp. 109-131.

[17] F. Belli, C. J. Budnik, "Test Minimization for Human-Computer Interaction," International Journal of Artificial Intelligence, Neural Networks, and Complex P, vol. 26, no. 2, Apr. 2007, pp. 161-174.

[18] F. Belli, A. Hollmann, "Test Generation and Minimization with 'Basic' Statecharts," Proc. of the 23rd ACM Symposium on Applied Computing (SAC 2008), ACM, Mar. 2008, pp. 718-723.

[19] A. C. Shaw, "Software Specification Languages Based on Regular Expressions," in Software Development Tools, W.E. Riddle and R.E. Fairley, Eds., Springer-Verlag, 1980, pp. 148-176.

[20] V. Bhattacherjee, D. Suri, P. K. Mahanti, "Software Testing: A Graph Theoretic Approach," Int. J. Inf. Commun. Techol., vol. 1, no. 1, Apr. 2007, pp. 14-25.

[21] F. Belli, J. Dreyer, "Program Segmentation for Controlling Test Coverage," Proc. of the 8th International Symposium on Software Reliability Engineering, Nov. 1997, pp. 72-83.

[22] F. Belli, K. E. Grosspietsch, "Specification of Fault-Tolerant System Issues by Predicate/Transition Nets and Regular Expressions - Approach and Case Study," IEEE Transactions on Software Engineering, vol. 17, no. 6, Jun. 1991, pp.513-526.

[23] R. David, P. Thevenod-Fosse, "Minimal Detecting Transition Sequences: Application to Random Testing," IEEE Transactions on Computers, vol. C-29, no. 6, Jun. 1980, pp. 514-518.

[24] F. Belli, M. Beyazıt, "Mutation of Directed Graphs - Corresponding Regular Expressions and Complexity of Their Generation,", Electronic Proc. in Theoretical Computer Science (EPTCS), vol. 3, J. Dassow, G. Pighizzini and B. Truthe, Eds., Jul. 2009, pp. 69-77.

[25] F. Bazzichi, I. Spadafora, "An Automatic Generator for Compiler Testing," IEEE Transactions on Software Engineering, vol. SE-8, no. 4, Jul. 1982, pp. 343-353.

[26] P. M. Maurer, "Generating Test Data with Enhanced Context-Free Grammars," IEEE Software, vol. 7, no. 4, Jul. 1990, pp. 50-55.

[27] E. G. Sirer, B. N. Bershad, "Using Production Grammars in Software Testing," Proc. of the 2nd Conference on Domain-Specific Languages (PLAN 1999), ACM, Oct. 1999, pp. 1-13.

[28] R. Lämmel, "Grammar Testing," Proc. of the 4th International Conference on Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol. 2029, H. Hußmann, Ed. Springer-Verlag, Apr. 2001, pp. 201-216.

[29] J. Offutt, P. Ammann, L. Liu, "Mutation Testing Implements Grammar-Based Testing," Proc. of the 2nd Workshop on Mutation Analysis (MUTATION 2006), IEEE CS Press, Nov. 2006, pp.12-12.

[30] A. Simão, J. C. Maldonado, R. da Silva Bigonha, "A Transformational Language for Mutant Description," Comput. Lang. Syst. Struct., vol. 35, no. 3, Oct. 2009, pp. 322-339.

[31] H. Zhu, P. A. Hall, J. H. May, "Software Unit Test Coverage and Adequacy," ACM Comput. Surv., vol. 29, no. 4, Dec. 1997, pp. 366-427.

[32] F. Belli, O. Jack, "Declarative Paradigm of Test Coverage," Software Testing, Verification and Reliability, vol. 8, no. 1, 1998, pp. 15-47.

[33] H. Li, M. Jin, C. Liu, Z. Gao, "Test Criteria for Context-Free Grammars," Proc. of the 28th International Computer Software and Applications Conference (COMPSAC 2004), IEEE CS Press, Sep. 2004, pp.300-305.

[34] P. Ammann, J. Offutt, Introduction to Software Testing, 2008.

[35] P. Purdom, "A Sentence Generator for Testing Parsers," BIT Numerical Mathematics, vol. 12, Apr. 1972, pp. 366-375.

[36] B. A. Malloy, J. F. Power, "An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases," Proc. of the 1st International Conference on Computer and Information Science, 2001.

[37] L. Zheng, D. Wu, "A Sentence Generation Algorithm for Testing Grammars," 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009), 2009.

[38] A. T. Acree, On mutation, Ph.D. thesis, Georgia Institute of Technology, 1980.

[39] F. Belli and C. J. Budnik, "Minimal spanning set for coverage testing of interactive systems," First International Colloquium on Theoretical Aspects of Computing (ICTAC 2004), Springer LNCS, vol. 3407, Sep. 2004, pp. 220-234.

[40] B. A. Malloy, J. F. Power, "A Top-down Presentation of Purdom's Sentence-Generation Algorithm," National University of Ireland Technical Reports, NUIM-CS-TR-2005-04, 2005.