UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

FAKULTÄT FÜR
ELEKTROTECHNIK,
INFORMATIK UND
MATHEMATIK

Exploiting Model Morphology for Event-Based Testing

Von der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

M.S. Mutlu Beyazıt

Erster Gutachter:	Prof. Dr.-Ing. Fevzi Belli
Zweiter Gutachter:	Prof. Dr.-Ing. Reiner Dumke

Tag der mündlichen Prüfung: 27.3.2014

Paderborn 2014

Diss. EIM-E/301

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

FAKULTÄT FÜR
ELEKTROTECHNIK,
INFORMATIK UND
MATHEMATIK

Zusammenfassung der Dissertation:

Exploiting Model Morphology for Event-Based Testing

des Herrn Mutlu Beyazıt

Testing is the process of checking a system under consideration whether it behaves as intended by the user. Model-based testing employs models for generation of test cases. Model-based mutation testing (MBMT) additionally involves fault models, called mutants, which are generated from the original model by applying mutation operators. A problem encountered in MBMT is caused by mutants that are equivalent to the original model (equivalent mutants) and multiple mutants that model the same faults. These mutants should be eliminated, because they lead to unnecessary test cases and thus increase the costs. Another problem of MBMT is the need to compare a mutant to the original model for test generation. Furthermore, using a single fixed model out of a set of various structurally, that is, morphologically, different models that describe a given system can also be considered as a problem of MBMT. This work proposes an event-based approach to MBMT that is not fixed on single events and a single model but rather operates on sequences of events of length $k \geq 1$ and invokes a sequence of models that are derived from the original one by varying its morphology based on the sequence length k. The approach employs formal grammars, introduces related mutation operators, and constructs algorithms, which enable the following and thus avoid the aforementioned drawbacks of MBMT: (1) the exclusion of mutants that are equivalent to the original model and multiple mutants that model the same faults; (2) the generation of a test case in linear time to kill a selected mutant without the necessity of 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 under consideration or studied in related literature. Three case studies validate the approach, analyze its characteristics and compare it to two other MBMT approaches and random testing. A discussion about the adaptation of the proposed MBMT approach using various models, its weaknesses and limitations, and further research potential in the field concludes the work.

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

FAKULTÄT FÜR
ELEKTROTECHNIK,
INFORMATIK UND
MATHEMATIK

Zusammenfassung der Dissertation:

Exploiting Model Morphology for Event-Based Testing

des Herrn Mutlu Beyazıt

Testen ist der Prozess zur Überprüfung eines betrachteten Systems, ob dieses sich wie durch den Benutzer vorgesehen verhält. Modell-basiertes Testen wendet Modelle zur Generierung von Testfällen an. Zur Testfall-Generierung benutzt modell-basiertes Mutation Testen (MBMT) zusätzlich Fehlermodelle, sogenannte Mutanten, welche durch die Anwendung der Mutationsoperatoren auf dem ursprünglichen Modell erzeugt werden. Ein Problem mit MBMT entsteht durch Mutanten, die äquivalent zu dem ursprünglichen Modell sind (äquivalente Mutanten) und mehrfache Mutanten, welche die gleichen Fehler modellieren. Diese Mutanten müssen eliminiert werden, sie verursachen unnötige Tests und erhöhen damit die Kosten. Ein weiteres Problem des MBMT ist die Notwendigkeit für die Testgenerierung, ein Mutant mit dem ursprünglichen Modell zu vergleichen. Außerdem kann die Verwendung eines einzigen festen Modells aus einem Menge von verschiedenen strukturell, d.h. morphologisch, unterschiedlichen Modellen, welche ein gegebenes System beschreiben, auch als ein Problem mit MBMT betrachtet werden. Diese Arbeit schlägt einen ereignis-basierten Ansatz für MBMT vor, der nicht auf einzelne Ereignisse und ein einziges Modell fixiert ist, sondern auf Sequenzen von Ereignissen der Länge $k \geq 1$ arbeitet. Dabei wird eine Sequenz von Modellen hergeleitet, die von dem ursprünglichen Modell durch Variation ihrer Morphologie basierend auf die Sequenzlänge k abgeleitet sind. Der Ansatz verwendet formalen Grammatiken, führt entsprechende Mutationsoperatoren ein und konstruiert Algorithmen, welche folgendes leisten und damit die o.g. Nachteile des MBMT vermeiden: (1) der Ausschluss der Mutanten, die äquivalent zu dem ursprünglichen Modell sind und der merhrfachen Mutanten, die die gleichen Fehler modellieren; (2) die Erzeugung eines Testfalls in linearer Zeit zum Eliminieren eines ausgewählten Mutanten ohne die Notwendigkeit eines Vergleichs mit dem ursprünglichen Modell; (3) die Analyse der morphologisch unterschiedlichen Modelle zur systematischen Erzeugung von Mutanten, wodurch die Menge der in der einschlägigen Literatur bekannten Fehlermodelle erweitert wird. Drei Fallstudien exemplifizieren den Ansatz, analysieren seine Eigenschaften und vergleichen ihn mit zwei anderen MBMT Ansätze sowie dem Zufallstesten.
Eine Diskussion über die Anpassung des vorgestellten MBMT-Ansatzes an verschiedene Modelle, seine Schwächen und Grenzen und weiteres Forschungspotential auf dem Gebiet schließt die Arbeit ab.

Contents

Symbols and Notation

Set Theory

$\{x_1, x_2, \ldots, x_n\}$	set of elements $x_1, x_2, \ldots, x_n$
$\varnothing$ or $\{\}$	the empty set
$x \in X$	x is an element of set X
$x \notin X$	x is not an element of set X
$\lvert X \rvert$	cardinality of set X
$X \cap Y$	intersection of set X and set Y
$X \cup Y$	union of set X and set Y
$X \subseteq Y$	set X is a subset of set Y
$X \setminus Y$ or $X - Y$	difference of set X from set Y
X'	complement of set X
$X.Y$ *or* XY	concatenation of set X and set Y
X^n	n-times concatenation of set X
X^*	closure of set X

Miscellaneous

$Q \rightarrow R$	production; Q can be replaced by R
$Q \mid R$	selection; Q *or* R
ε	the empty string
$\lvert s \rvert$	length of string/sequence s
$Q \Rightarrow_G^* R$ or $Q \Rightarrow^* G$	derivation; Q can be derived from R using one or more productions of grammar G (in the latter, G is implicit)
$Q \Rightarrow_G R$ or $Q \Rightarrow R$	derivation step; Q can be derived from R using one production of grammar G (in the latter, G is implicit)

$L(X)$	language defined by abstraction X
$r = r_1\ r_2\ \ldots\ r_k$	k-sequence r
$c(r)$	context of k-sequence r
$[$	start node
$]$	end node
(q, r)	arc; from q to r
$d(s)$	corresponding basis sequence of sequence s; decontexted version of s
$d(X)$	corresponding set of basis sequence of set of sequences X
$u^1\ \ldots\ u^m$	sequence of m k-sequences; sometimes, m-derived sequence in a k-Reg
$T_P(G)$	set of all positive test cases of grammar G
$T_{CES}(G)$	set of all complete event sequences of grammar G
$T_N(G)$	set of all negative test cases of grammar G
$T_{FCES}(G)$	set of all faulty complete event sequences of grammar G
$T_S(s, k)$	sequence transformation of s based on integer k
$T_S^{-1}(s, k)$	inverse sequence transformation of s based on integer k
$R+P$	R or P; union of regular expressions R and P
$R.P$ or RP	concatenation of regular expressions R and P
R^*	(star) closure of regular expressions R
R^+	$R.R^*$
$\sqsubseteq$	risk ordering relation
$\sqsubseteq_h$	holistic risk ordering relation
$rl(s)$	risk level of state s
$T(p, a, X)$	PDA transition function
δ	quiescence
θ	time
$p(x)$	push
$q()$ or $q(x)$	pop
$r()$ or $r(x)$	read

Mutation Operators

Ms	mark start (for a k-Reg)
Mf	mark finish (for a k-Reg)
Mns	mark nonstart (for a k-Reg)
Mnf	mark nonfinish (for a k-Reg)
Is	insert sequence (for a k-Reg)
It	insert terminal (for a k-Reg)
Os	omit sequence (for a k-Reg)
Ot	omit terminal (for a k-Reg)
Msti	mark state initial (for a PDA)
Mstf	mark state final (for a PDA)
Mstnf	mark state nonfinal (for a PDA)
Msyi	mark stack symbol initial (for a PDA)
Itr	insert transition (for a PDA)
Ist	insert state (for a PDA)
Otr	omit transition (for a PDA)
Ost	omit state (for a PDA)
Rw	write replacement (for transition of a PDA)
Rw-read	replace with read (for transition of a PDA)
Rw-push	replace with push (for transition of a PDA)
Rw-pop	replace with pop (for transition of a PDA)
Rw-poppush	replace with pop-push (for transition of a PDA)
Rr	read replacement (for transition of a PDA)
Rr-init	replace with initial stack symbol (for transition of a PDA)
Rr-top	replace with new stack top (for transition of a PDA)
Rr-another	replace with another stack symbol (for transition of a PDA)
Re	event replacement (for transition of a PDA)
Rs	source replacement (for transition of a PDA)
Rd	destination replacement (for transition of a PDA)
Ito	insert token (for a PN)
Ia	insert arc (for a PN)
Itr	insert transition (for a PN)

Ip	insert place (for a PN)
Oto	omit token (for a PN)
Oa	omit arc (for a PN)
Otr	omit transition (for a PN)
Op	omit place (for a PN)

Abbreviations

CES	complete event sequence
CFG	context-free grammar
EIG	event interaction graphs
ESG	event sequence graph
ESG4WSC	event sequence graphs for web-service compositions
ESIG	event semantic interaction graphs
ESOR	expected state omission ratio
EFG	event flow graph
FCES	faulty complete event sequence
FSCR	fault state coverage ratio
FSIR	fault state inclusion ratio
FSA	finite state automaton / finite state automata
FSM	finite state machine
GUI	graphical user interface
k-Reg	k-sequence right regular grammar
MBT	model-based testing
MBMT	model-based mutation testing
PDA	pushdown automata
PEFG	probabilistic event flow graph
RE	regular expression
RG	regular grammar
SUC	system under consideration
SUT	system under test
USIR	unexpectd state insertion ratio
UML	unified modeling language

Part I

Introduction

1 Introduction

Testing is a user-centric quality assurance technique based on *test cases* that consist of *test inputs* and expected *test behaviors* (commonly characterized by *test outputs*). A *test* invokes the execution or training of the *system under consideration (SUC)* using a test case; that is, a *test path* is exercised in the *test object* using a test case. SUC *passes* the test if, upon a test input, the expected behavior is produced; otherwise, the SUC *fails* the test. This entails the tough *oracle problem* [85] for deriving the expected behavior to determine the correctness of the observed 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* (also called *coverage metric*) [178] is used as a stopping condition for testing and providing a measure of the quality of a test set. This work 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 [24]. 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 [25]. The use of models has various advantages, such as increasing effectiveness and efficiency in terms of fault detection and costs [93]. Formal models additionally help to avoid the oracle problem in the sense that the expected test outputs can automatically be generated [93][165]. Also, contrary to the common belief, model-based approaches are not that hard to learn and apply [179].

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 [76]. Therefore, expressiveness is a major factor which influences effectiveness of a testing approach in terms of fault detection and costs. In general, as expressiveness increases, analyzability decreases [80]. Hence, the use of models with insufficient expressiveness may cause a decrease in

the fault detection effectiveness; whereas, the use of models with excessive expressive power may cause an unnecessary decrease in the cost effectiveness.

Some models have the same expressiveness; classical examples are finite state automata (FSA), regular expressions (REs), and regular grammars (RGs) [100], as they relate to the same class of formal languages, that is, type-3 languages [54]. A substantial amount of work in practice relies on the use of such models. Also, finite state machines [123][128] and pushdown automata (PDA) [100] are examples of test models which are more expressive than FSA; whereas event sequence graphs (ESGs) [25] are less expressive. Statecharts [87] and UML diagrams [133][70], on the other hand, do not have formally defined semantics [88][23][56]; that is, they are not completely formalized. Depending on how the semantics are defined, they may possess more expressiveness than FSA models. Generally, all these models are used in MBT. However, models which have similar expressiveness to FSA are especially preffered for MBMT; because such models possess better analyzability; for example, the equivalence problem is decidable.

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) [171], on the other hand, are event-based [39][35]; 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 work is event-based. In the context of this work, 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 [35]. This is one of the reasons why this work 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; that is, the test object behaves correctly. 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, because a different test path is exercised in the test object. *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 work does not use the term “structure” to avoid a possible confusion with the term “structural testing” [132].) In MBT, the differences in morphology may cause the associated fault models and the generated test sets to be different.

The *model morphology* this work 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; therefore, they can be used to exercise different test paths in the test object by generating test cases possessing different characteristics. This way of model exploitation differs principally from the existing ones; for example, the one used by Unified Modeling Language (UML), which creates different kinds of models (diagrams) for different views [149].

Model-based mutation testing (MBMT) [52][8][40] 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* or *discriminate* 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; that is, the test case aims to check whether or not a specific fault exists on a certain test path in the test object. Evidence demonstrates that using such model-based mutants is effective at detecting both code-based mutants and real-world faults [10][19].

MBMT has problems similar to those of (code-based) mutation testing [59][86][4] 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 [5]. Grün, Schuler, and Zeller, among other authors, [84] 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 [134][26][29]. This work introduces a new approach based on 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. Thus, one does not generate multiple test cases that check existence of the same fault on a certain test path in the test object.

- **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. Hence, one can efficiently check whether or not there exists a certain fault on a specific path in the test object.
- **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. In this way, one can incrementally model different or more subtle faults that are located on different test paths in the test object. Existing approaches do not consider such fault models.

As mentioned above, the approach proposed in this work provides significant improvements to the existing event-based MBMT approaches [40][26][99]. These improvements are not based on system-specific semantics because such semantics generally requires an access to the internals of the system and limits the generality of the approach. Therefore, the proposed approach can be applied to all systems for which an event-based model is feasible to use. For example, event-based approaches are often used for testing of desktop applications [39][27], embedded systems [40], web-based systems [43][26][46][28][30] and graphical user interfaces [124]. This work uses web-based and embedded systems that possess different characteristics to demonstrate the improvements of the proposed approach.

It is also possible to use the proposed approach for both verification and validation purposes, considering the fact that testing (more specifically, model-based or functional testing) is regarded as an activity in both verification and validation [103]. For verification, one can construct a model from a given system specification to determine whether the system behaves as described in this specification. For validation, a model needs to be constructed to describe the user requirements or functionality so that one can evaluate whether the system behaves as intended by the user or the customer. In this work, the models used in case studies are constructed from (informal) system specifications that are prepared for development purposes. Therefore, MBT is performed as a verification activity, assuming the correctness of the system specifications.

The work is organized in three parts. Part I serves for an introductory purpose; Chapter 2 explains the basic idea behind the main approach presented in this work by way of an example and Chapter 3 discusses the related work.

In Part II, the approach is introduced and evaluated in detail. The basic notions related to the approach are given in Chapter 4. Chapter 5 introduces the concepts related to variation of model morphology to extend the set of faults models and to generate test cases, and Chapter 6 defines and discusses event-based mutation operators to generate mutants from the morphologically different models. Chapter 7 discusses strategies for mutant selection from the obtained morphologically different models for the purpose of test generation and outlines mutant-based test generation aspects. Chapter 8 performs three case studies to analyze the characteristics of the approach in comparison to random testing approach, ESG-based MBMT approach (which employs coverage-based test generation from mutants), and *mutate-and-kill-based (MK-based)* MBMT approach (which is based on the idea of generating discriminating test cases by analyzing or comparing mutants against the original model).

Part III concludes the work. Chapter 9 discusses some further perspectives related to the application of the proposed approach for vulnerability testing, the adaptation of the MBMT approach using different models and the extension of the event-based models. Chapter 10 concludes the work and outlines some future research.

2 Basic Idea Demonstrated by an Example

This chapter sketches the approach by means of a simple example. The next three sections explain usage of formal grammars for modeling, mutant generation, and grammar transformation for varying the model morphology. Novelties are exemplified in the last section.

Example 2.1 (Running Example). Consider three events

c: copy, *x: cut* and *p: paste*.

For simplicity, events to *select* and *deselect* system objects or locations are ignored, assuming that these events are performed properly before any events from *{c, x, p}*.

- At the beginning, one can perform either *c* or *x*.
- *c* can be followed by either *c*, *x* or *p*.
- *x* can be followed by either *c*, *x* or *p*.
- If *p* is performed after *c*, it can be followed by either *c*, *x* or *p*.
- 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 performing a *p*.

2.1 Event-Based Modeling Using Grammars

Figure 2.1a represents an event-based directed graph model to illustrate Example 2.1. Such models are popular in the testing community [35] and have the same expressiveness as FSA (or FSA with outputs, if output events are also included). Since events are the observable entities in model-based testing and states generally represent the internals of the system, the proposed model focuses on events and refrains from visualizing states. Therefore, events are placed at the

nodes, and the *follows* relation between the events is described using arcs. Pseudo-events *[* and *]* are used to mark, respectively, the start and finish events [25].

(a) A model containing ambiguity.

(b) A model with contexted events (Ambiguity is removed by indexing/contexting).

Figure 2.1. Event-based models for Example 2.1.

The model in Figure 2.1a has a severe drawback. By "event *p*," one cannot differentiate to which *p* event is referred. The present approach suggests distinguishing such events from each other by indexing that considers the *context*s in which they reside, leading to *contexted events*, such as *{c1, x1, p1, p2}* (Figure 2.1b). Their counterparts, *basis events*, such as *{c, x, p}*, represent the events as they are visible to the user. Note that contexted events are not necessarily caused by cycles or loops in the model.

Fault models associated with the models like the one in Figure 2.1b are primarily based on modifying the *follows* relation between single events. This modification needs to be generalized by analyzing occurrences of single events with respect to event sequences of length $k \geq 1$ (*k-sequences*) for systematic extension of event-based fault modeling. In this way, one can incrementally model different or more subtle faults that are on different test paths in the test object.

Grammars are suitable for representing event-based abstractions based on k-sequences. They allow multiple occurrences of events in *productions*, which enable to represent the *follows* relation between k-sequences and events. This practice is common in compiler construction and testing [100]; related techniques are exploited here.

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

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

Figure 2.2. A grammar model for Example 2.1 which makes use of 1-sequences.

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

- *c(a)* → *b c(b)* means that *b* follows *a* and *a b* is a 2-sequence.
- *S* → *a c(a)* means that *a* is a start event.
- *c(a)* → ε means that *a* is a finish event.

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

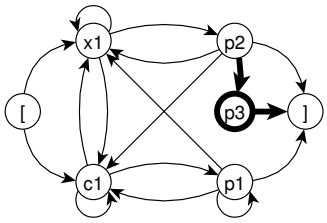
The productions of Figure 2.2 form a regular grammar. The terminals therein are events that can be viewed as 1-sequences, and the nonterminals are contexts. Therefore, this model is called "1-Reg."

2.2 Generating Mutants

The new approach refines the elementary mutation operators *insertion* and *omission* [40] to modify sequences of events by also considering the *start* and *finish* events. The iterative and combinatorial deployment of these operations enables further mutation operators such as *duplication*, *deletion*, or *replacement* [134][18].

Example 2.3 (Mutants). Figure 2.3 contains some mutants of Example 2.1. The mutant in Figure 2.3a is generated using an event-based mutation [40] by inserting event/terminal *p3*. Furthermore, the mutant in Figure 2.3b is generated using a grammar-based mutation [134][18] by replacing terminal *p1* by *x1*.

These mutants are different; the modeled faults are on different test paths in the test object. The mutant in Figure 2.3a is a 1-Reg; it models a single fault: "*p* is extra after *x p*." In contrast, the mutant in Figure 2.3b is not a 1-Reg but an RG; it models multiple faults: "*p* is missing after *c*," and "*p* is extra after *c x p*."



(a) An insert event mutant.

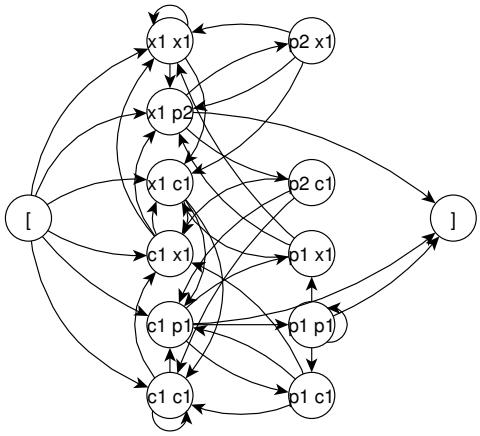
S → c1 c(c1) | x1 c(x1)
c(c1) → c1 c(c1) | x1 c(x1) | **x1** c(p1)
c(x1) → c1 c(c1) | x1 c(x1) | p2 c(p2)
c(p1) → c1 c(c1) | x1 c(x1) | p1 c(p1) | ε
c(p2) → c1 c(c1) | x1 c(x1) | ε

(b) A terminal replacement mutant.

Figure 2.3. Some mutants of the model in Figure 2.2 (Mutations are shown in boldface or underlined).

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

(a) Productions.



(b) Directed graph visualization.

Figure 2.4. A grammar model for Example 2.1 which makes use of 2-sequences (Transformed from Figure 2.2).

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 Figure 2.2 and its transformation shown in Figure 2.4 describe the same system behavior, but productions in Figure 2.4 utilize 2-sequences; therefore, it is a "2-Reg." A 2-Reg is morphologically different from its 1-Reg, for example, production of the form $c(a\ e) \rightarrow e\ b\ c(e\ b)$ means that *b* follows *a e* and *a e b* is a 3-sequence.

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 Figure 2.4 generated by omitting sequence *(p1 c1, c1 p1)* as shown in Figure 2.5b. This mutant models the fault that

p1 is missing after *p1 c1*;

that is, paste fails after performing a paste and a copy. It is not possible to create such a mutant from the model in Figure 2.2 by a simple omission. For example, one can omit sequence *(c1, p1)* (See Figure 2.5a). However, in this mutant, paste fails immediately after performing a copy. Hence, the mutant in Figure 2.5b models a different and more subtle fault than the mutant in Figure 2.5a. Thus, the set of fault models can be extended by generating mutants modeling different or more subtle faults; that is, one can incrementally model different or more subtle faults that are located on different test paths in the test object.

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 [17][8], 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.

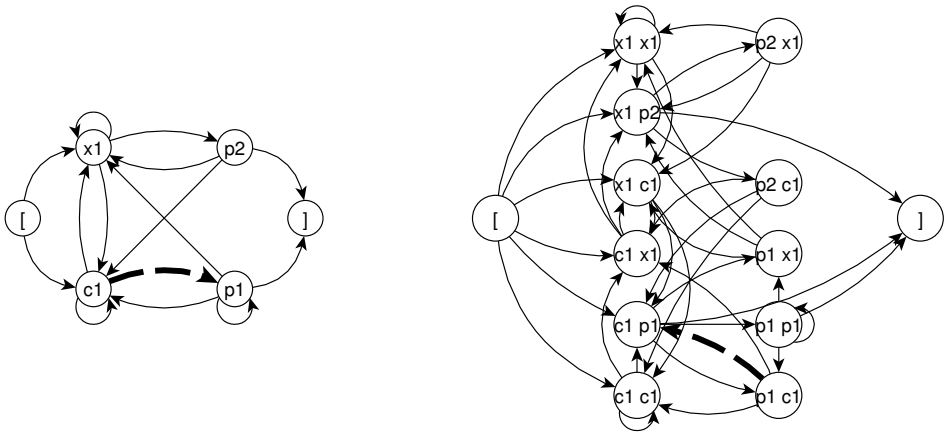
Each selected mutant has the following properties. (1) It does not violate the type-3ness of the given grammar; that is, the mutated grammar is of the same type as the original one (Also, see the discussion in Section 3.4). (2) It models a small number of faults. (3) The faults are located at the mutation point; that is, the faults

are directly related to the mutation parameter. Doing so, one avoids modeling the same fault that is located on a certain test path in the test object multiple times.

The mutant in Figure 2.3a 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 Figure 2.3b is excluded because it models multiple faults which can be modeled separately. *p* is missing after *c*, and *p* is extra after *c x p*.

A test case is generated in linear time to kill a mutant. Since the location of the faults modeled by each selected mutant can be determined from the actual mutation parameter, a unique test case to kill the mutant can be generated in linear time, without comparing it against the original model. Hence, one can efficiently check whether or not there exists a specific fault on a certain path in the test object or not. For example, breadth-first search can be used to generate *x1 p2 p3* to kill the mutant in Figure 2.3a.



(a) A mutant of the model in Figure 2.2.

(b) A mutant of the model in Figure 2.4.

Figure 2.5. Two mutants for Example 2.1 (Mutations are shown in boldface dashed lines).

3 Related Work

Related work is discussed in the relevant categories as follows.

3.1 “Transformation” and “Mutation”

The grammar transformation is used in this work for varying the morphology of a given model. This should not be confused with similar notions used in other approaches, such as model transformation [125], input/output transformation in metamorphic testing [177], and model composition [113] 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 behavior using k-sequences and another one describing it 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 work 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 [25].

In aspect-oriented modeling [113], 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 work.

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 [127]. In this work, 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.

3.2 Model-Based Mutation Testing

Mutation testing (or mutation analysis) is a method to test a given system by making use of systematically generated mutants, which represent faulty versions of the system. It was originally introduced as a code-based approach based on certain assumptions about the developer and the faults [59][86][4][3]. However, it can also be applied to different formalisms, that is, models (or specifications) [52][81].

Basically, mutation testing can be regarded both as an evaluation technique (to assess the fault detection adequacy of a given test set by measuring its ability to detect the faults seeded into the mutants) [6] and as a testing technique (to improve the testing process by using the mutants) [175][89]. 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 to decide whether or not a given mutant is equivalent to the original system, certain equivalent mutants can be detected [136] and specific techniques such as program slicing can help to reduce the effort involved in equivalent mutant detection [94]. However, the problem of generating multiple mutants which are equivalent to each other is not considered. Furthermore, only a fixed program and its mutants are utilized in mutation testing, limiting the set of faults especially while using mutation testing as a testing technique.

In classical sense, mutation testing is performed on an implementation to test the implementation itself [6][58][107]. Similarly, it is also applied to a model to test the model itself [158][71][73]. In a non-classical sense, mutation testing is used to test an implementation based on its model [52][81][8][40], which is referred to as *model-based mutation testing (MBMT)*. (For a detailed survey of mutation testing, reader can refer to [104].)

In MBMT, mutants of a model have morphological differences; however, the primary purpose is to model faults drawn from practice [52][81][8][40]. This

work 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 work.

Stocks [156] discusses MBMT for Z specifications. He discusses different types of mutations for Z and defines criteria to choose test cases to distinguish certain types of mutants from the original model.

Amman, Black, and Majurski [17], and Black, Okun, and Yesha [47][48] 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 [137][49][79].

Kovács, Pap, Viet, Wu-Hen-Chang and Csopaki [112] apply MBMT using SDL. They define six mutation operators and two algorithms for test selection using mutants to automate the process. Only the first algorithm is actually related to MBMT, where a mutant is compared to the original model using a state space exploration algorithm and a test case revealing the inconsistency between the two models is generated. The second algorithm is used to select an optimal subset of a given test set so that the achieved mutation score does not change.

Aichernig [8], and Aichernig and He [11] develop a theory based on the notion of refinement, which is applied using different types of abstractions in practice, such as language of temporal ordering specification for testing protocols [167] and action systems for testing embedded systems [9], some of which may contain nondeterminism. The idea is similar to the above. However, since an equivalence check is too strong and results in too many test cases, a refinement check is used for conformance, such as input output conformance [163]. In this way, mutants that are not equivalent but conform to the original model are also discarded. Improvements to the refinement checking are performed [12][13][14][15].

Belli, Budnik, and Wong [40] 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 [26][36][37][161][99]. In these works, equivalent mutants are not really excluded and used in coverage-based test generation to populate the test set, because test cases generated from them can still reveal additional faults even if they do not contribute to the intended coverage [77][99]. Also, although some works [40][26][99] adopt the transformation defined by Belli and Budnik [38][39], 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.

The approach proposed in this work 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 [134][18], 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 [17][8], 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, which do not exercise any faulty behavior, when compared to [40][26][99].

3.3 Fault Domain-Based Testing

Fault domain-based testing is based on defining a fault domain, that is, a set of faults that can be inserted into the model (or the specification) and are intended to be discovered. In general, finite state machines (FSMs) are used. Under certain conditions, test sets generated using these approaches are proved to reveal of all possible faults from the defined domain. The proofs are based on the relationship between the specification and the implementation. In other words, if the required conditions are satisfied, then the generated test set is capable of proving the equivalence or trace inclusion (depending on the chosen conformance relation) between specification and implementation.

Most of the well-known fault-based testing methods generally require a subset of the following assumptions to hold.

- Fault domain: Only certain types of faults are considered.
- Reliable reset: There exists a reliable operation that brings the implementation to its initial state.
- FSM implementation: The implementation can be represented/abstracted by an FSM model.
- Number of states in the implementation: Prior to generating the tests, an upper bound on the number of states in the implementation is known by the tester.
- Connectedness: FSM is strongly connected or initially connected.

- Determinism: FSM is deterministic or non-deterministic.
- Reducedness: (Deterministic) FSM is reduced or non-reduced.
- Completeness: (Deterministic) FSM is complete or partial.

Fault domain-based method can be classified into two categories [141]: (1) Method which yields a single test case; also called checking sequence [83][91][95]. (2) Methods which yields multiple test cases such as W [55][166], Wp [78], HSI [118], SC [142], H [61], SPY [153] and P [154].

For example, all the methods in (2) above assume that an upper bound on the number of states in the implementation is known, and the specification is initially connected and deterministic. W, Wp and SPY methods additionally assume that FSM is reduced and complete; whereas, HSI, H and P methods assume that it is reduced but can be partial. SC is the only method that can also work with non-reduced and partial specifications. Another fundamental difference among these methods is the total length of the genetrated test cases. Researchers investigate to reduce the test set size, while keeping the same test properties. Experimental studies comparing these methods are also performed [60][68][69].

While employing fault domain-based testing methods in practice, problems may arise. The required assumptions may be hard to satisfy or handle for testing of certain real-world systems. For example, it is not very usual for one to approximate a good upper bound on the number of implementation states in model-based testing, because system internals are not available. However, when the approximation is not well enough, the size of generated test set increases and the testing efficieny decreases. Even if the required assumptions are feasible, fault-based testing methods tend to generate test sets having relatively much larger sizes, when compared to, for example, coverage-based test generation methods. Also, the methods which rely on weaker assumptions, such as SC, are quite impractical and do not scale well (even when compared to other fault domain-based methods). Furthermore, the fault domains assumed by these methods contain only the faults that can be revealed using some positive test cases; that is, the extra event faults considered in this work are not considered by fault domain-based approaches.

Note that the above discussion primarily focuses on the fault domain-based testing methods which assumes deterministic FSMs, similar arguments holds for the methods that work with nondeterministic FSMs, such as the ones proposed by Yevtushenko et al. [172][173], Aboelfotoh et al.[1], Kloosterman [110], Petrenko et al. [144][143][141], Tripathy et al. [164], Alur et al. [16], Hierons [92], and Hwang et al. [101]. Due to the nature of the work, the required assumptions tend to become harder to satisfy for increasing number and more varied type of real-life systems.

3.4 Grammars in Testing

Software testing practice contains a substantial amount of work based on grammars for generating well-formed inputs [146][122], testing interpreters [155], and, in general, testing software termed as *grammarware*, such as compilers, debuggers, code generators, and documentation generators [109] using different grammar-based formalisms, for instance, definite clause grammars [139], attribute grammars [138] and graphs grammars [67]. 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.

S → deposit ACC0 | debit ACC0
ACC0 → digit ACC1
ACC1 → digit ACC2
ACC3 → digit AMM0
AMM0 → $ AMM1
AMM1 → digit AMM2
AMM2 → digit AMM2
AMM2 → . AMM3
AMM3 → digit AMM4
AMM4 → digit ACT | digit S
ACT → ε

(a) RG.

S → deposit ACC0 | debit ACC0
ACC0 → digit ACC1
ACC1 → digit ACC2
ACC3 → digit AMM0
AMM0 → $ AMM1
AMM1 → digit **AMM2 AMM2**
AMM2 → digit AMM2
AMM2 → . AMM3
AMM3 → digit AMM4
AMM4 → digit ACT | digit S
ACT → ε

(b) CFG.

Figure 3.1. An RG and its nonterminal duplication mutant (drawn from [18]).

In this respect, the approach in this work contains similarities with an existing approach [134][18] that also uses grammar-based models and mutation operators. However, the present work 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 [100]. This drawback is exemplified using the regular grammar in Figure 3.1a that is drawn from an example used by Amman and Offutt [18], and slightly, nevertheless equivalently, reformatted for saving space. The nonterminal duplication mutant in Figure 3.1b is a CFG.

3.5 An Overview of Different Test Models

There are plenty of models used in practice which have various degrees of expressive power and possess different semantics.

Finite state machines [123][128] and timed input/output automata [106] are extended versions of FSA and include explicit system outputs (timed input/output automata also include time parameter). These formalisms have been used, for example, for modeling role-based [120] and timed role-based [121] access control policies and to generate test cases.

Pushdown automata [100] can also be considered as FSA with additional stack components to store some elements during the computation. Thus, they have more expressive power than FSA. These models are also used for modeling and testing of specific software functions [159][36][37].

Statecharts [87] are an extended form of finite state machines designed to capture hierarchy and concurrency. There are a variety of statecharts with varying execution semantics [23].

Process algebras such as communicating sequential processes [98][57] and calculus of communicating systems [126] are developed to model, study, and test the systems of concurrent, communicating components.

Petri nets [148] are formal models that can be considered as an extension of FSA where the notions of “transitions” and “states” are made explicitly disjoint. They are used, for instance, to model the behavior of concurrent systems, including synchronization of processes.

Unified modeling language (UML) [133][70] enables the designer to describe a system at different levels of abstraction by means of a set of diagrams. In order to specify a system in an easier way, UML is composed of less precisely defined visual formalisms called UML diagrams, such as state machine diagrams and sequence (or event) diagrams. These diagrams have no exact semantics [56]; by defining the semantics properly, they can still be used in testing [108][50].

This work employs an MBMT approach which makes use of an event-based RG model. The reasons for selecting this model for MBMT, not the models discussed above, are outlined in the following.

As explained in Section 2.1, the proposed approach aims to use an event-based model which makes use of event sequences of length k and the relation between these sequences and events. RGs seem to be the most appropriate choice, because they allow multiple occurrences of events in productions and equivalence of two RGs is decidable [100]. Furthermore, the expressiveness of this grammar model is as strong as the most of the models that are often used in MBT (and MBMT), such as FSA and FSMs (if outputs are included).

One can also consider event diagrams (or sequence diagrams) of UML as an appropriate choice for event-based modeling. These diagrams are generally used to model certain scenarios in the form of interactions between different system processes or objects. Therefore, they also contain events which are not visible at user level. Thus, one needs internal information regarding the system to construct event diagrams properly. Furthermore, these diagrams contain no mechanism to model using event sequences of length k.

In addition, there are two major reasons that this work avoids from using models such as timed automata, pushdown automata, statecharts, process algebras and Petri nets. First, the approach proposed in this work does not include aspects like time, memory, communication and concurrency; it concentrates on other but novel aspects: varying the model morphology, exploiting the morphological differences and developing mutant selection strategies. Still, the proposed event-based model is informally extended in Section 9.3 to include additional aspects and lay some ground for further future research.

Second and more importantly, MBMT approach, which is adapted and used in this work, relies on the comparison of each generated mutant to the original model to check for equivalence. In general, the equivalence is not decidable for models such as Petri nets [97] and pushdown automata [100]. This decreases the performance of MBMT significantly. Furthermore, it causes the proposed approach to lose its major benefits discussed in Section 2.4. Still, the use of models like pushdown automata and Petri nets for MBMT is considered as a further aspect in Section 9.2 and some initial steps are taken for an adaptation.

Part II

Approach and Case Studies

4 Notions Used

The goal of this chapter is to show how formal grammars can be used to perform event-based modeling and give the related terminology.

Definition 4.1 (Grammar). A *formal grammar* (or just *grammar*) is a 4-tuple $G = (N, E, P, S)$ where

- N is a finite set of *nonterminal symbols* (or *nonterminals*),
- E is a finite set of *terminal symbols* (or *terminals*),
- P is a finite set of *production rules* (or *productions*) of the form

  $$Q \rightarrow R$$

  where $Q \in (N \cup E)^* N (N \cup E)^*$ is the *head* of the production, $\rightarrow$ is the *production symbol* and $R \in (N \cup E)^*$ is the *body* or *tail* of the production, and
- $S \in N$ is a distinguished nonterminal *start symbol*.

Working with grammars, the following notions are used. Given a grammar $G = (N, E, P, S)$. Productions are used in generation or derivation of strings. A *derivation*, denoted by $\Rightarrow_G^*$ is a sequence of *derivation steps* each of which is of the form $xQy \Rightarrow_G xRy$ where $x,y \in (N \cup E)^*$ and $Q \rightarrow R \in P$ ($\Rightarrow^*$ and $\Rightarrow$ are used when there is no confusion). The number of derivation steps in a derivation is called *the length of the derivation*. Since a grammar is used to generate a language, the *language* defined by grammar G is the set of strings $L(G) = \{w \mid S \Rightarrow^* w \ (w \in E^*)\}$ and each string in $L(G)$ is called a *sentence* of G.

Most of the model-based approaches make use abstractions which are based on FSA (or FSA with outputs). From the Theory of Automata and Formal Languages, FSA accept type-3 languages that are generated by regular grammars (RGs). Hence, FSA are recognizing devices while RGs are generative. Following, RGs are introduced with the goal to transfer the model in Figure 2.1 into an equivalent grammar and define an event-based RG model.

Definition 4.2 (Regular Grammar (RG)). Given a grammar $G = (N, E, P, S)$. G is said to be a *left regular grammar* if its productions are in one of the following forms:

$$Q \to \varepsilon,\ Q \to x, \text{ or } Q \to Rx,$$

and it is said to be a *right regular grammar* if its prodcutions are in one of the following forms:

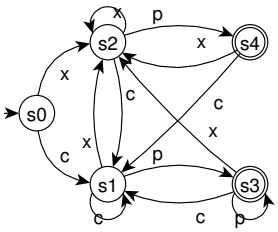
$$Q \to \varepsilon,\ Q \to x, \text{ or } Q \to xR,$$

where $x \in E$, and $Q,R,S \in N$, and ε is the *empty string*. A *regular grammar (RG)* is a formal grammar which is either left regular or right regular. If x is allowed to be a (possibly empty) sequence of terminals ($x \in E^*$) then the grammar is said to be an *extended* (*left* or *right*) *regular grammar*.

Example 4.1 (An RG and an FSA). Figure 4.1 shows an RG model for Example 2.1 which is equivalent to the event-based model in Figure 2.2 can be transferred into an equivalent RG as shown in Figure 4.1.

S → c A | x B
A → c A | x B | p C | p
B → c A | x B | p D | p
C → c A | x B | p C | p
D → c A | x B

(a) RG.



(b) FSA visualization.

Figure 4.1. An RG model for Example 2.1.

FSA and RGs are not purely event-based. Furthermore, FSA do not capture the system behavior using event-sequences of certain length. Thus, in the light of the discussion in Section 2.1 (and above), the following event-based RG model is proposed for event-based modeling.

Definition 4.3 (k-Sequence Right Regular Grammar (k-Reg)). A *k-sequence right RG (k-Reg) (integer $k \geq 1$)* is a 6-tuple $G = (E, B, K, C, S, P)$ where:

- E is a finite set of *events* (or *contexted events*).
- B is a finite set of *basis events*, which is the set of all visible events under consideration. For each event $e \in E$, $d(e) \in B$ is the corresponding basis

event, which is the noncontexted version of e, and $d(.)$ is the *decontexting function*.

- $K \subseteq E^k$ is a finite set of *k-sequences* (or *terminals*). For each k-sequence $r \in K$, $r = r_1 \ldots r_k$ and $d(r) = d(r_1) \ldots d(r_k) \in B^k$ is the corresponding *basis k-sequence*.
- C is a finite set of *contexts* (or *nonterminals*) $S \in C$ *is the start context (or start symbol)*.
- P is a finite set of *productions* of the form

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

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

  $$q_2 \ldots q_k = r_1 \ldots r_{k-1},$$

  that is, ending (k-1)-sequence of q is the beginning (k-1)-sequence of r.

Note that k-sequences are defined as terminals, and they have different, therefore, unique contexts. The semantics of the productions of a k-Reg G is as follows.

- For each $c(q) \rightarrow r\ c(r) \in P$, where $q = q_1 \ldots q_k$ and $r = r_1 \ldots r_k$, r *follows* q *in grammar* G, and r_k *follows* q *in the system modeled by grammar* G; that is, $q_1 \ldots q_k\, r_k$ is a (k+1)-sequence in the system.
- For each $S \rightarrow r\ c(r) \in P$, r is a *start k-sequence*.
- For each $c(q) \rightarrow \varepsilon \in P$, q is a *finish k-sequence*.

These productions allow only right linearity, ensuring type-3 preservation.

Using such semantics allows one to encode (k+1)-sequences using the productions of k-Regs. To see this, for a given k-Reg G, let $c(q) \rightarrow r\ c(r) \in P$, where $q = q_1 \ldots q_k$ and $r = r_1 \ldots r_k$, then:

- If G is a 1-Reg, the production represents a 2-sequence

  $$q\ r = q_1\ r_1$$

  in the system modeled by G.
- If G is a 2-Reg, the production represents a 3-sequence

  $$q\ r_2 = q_1\ q_2\ r_2 = q_1\ r_1\ r_2 = q_1\ r$$

  in the system modeled by G (because $q_2 = r_1$).
- If G is a 3-Reg, the production represents a 4-sequence

  $$q\ r_3 = q_1\ q_2\ q_3\ r_3 = q_1\ r_1\ r_2\ r_3 = q_1\ r$$

  in the system modeled by G (because $q_2\ q_3 = r_1\ r_2$).

Thus, in general, if G is a k-Reg, the production represents a (k+1)-sequence

$$q\ r_k = q_1 \ldots q_k\ r_k$$

in the system modeled by G (because $q_2 \ldots q_k = r_1 \ldots r_{k-1}$ for $k \geq 2$). Such sequences are also supposed to be in the SUC or the test object.

Usually, a grammar (Definition 4.1) is a 4-tuple: A set of nonterminals, a set of terminals, a set of productions and a start symbol. Also, events are generally treated as the terminals. However, in the case of k-Regs (Definition 4.3), terminals are defined to be k-sequences and events are included additionally to focus on the relation between the k-sequences (and the events). Events are included, because they are the building blocks of the k-sequences and the relation between the k-sequences is affected by these events. Since events of a k-Reg are contexted, a set of basis events is also included to preserve the association between contexted events and basis events, because, for example, one needs to replace contexted events in a set of test cases by the basis events for test execution.

Productions of a k-Reg can be visualized using directed graphs (which can be considered as a variant of ESGs).

- Nodes are labeled using the k-sequences in K, and *[* and *]*.
- Arcs correspond to the productions in P.
    - Arcs of the form *([, r)* correspond to the productions of the form $S \rightarrow r\ c(r)$.
    - Arcs of the form *(r,])* correspond to the productions of the form $c(r) \rightarrow \varepsilon$.
    - Arcs of the form "*(q, r)* correspond to the productions of the form $c(q) \rightarrow r\ c(r)$.

Such visual representations are helpful in understanding. Therefore, they are often used to graphically represent the productions of the k-Reg models in the rest of the discussion.

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

Example 4.2 (A 1-Reg). The following 6-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 $p = d(p1) = d(p2)$.
- $K = E$, because $k = 1$.
- $C = \{S, c(c1), c(x1), c(p1), c(p2)\}$.
- S is the start context.

- *P* contains *15* productions (See Figure 2.2 or Figure 2.1b).

In Figure 2.2, *c1* and *x1* are start 1-sequences (or events) which are marked by the productions of the form $S \rightarrow e\ c(e)$ (or arcs of the form *([, e)*). Also, *p1* and *p2* are finish 1-sequences which are marked by the productions of the form $c(e) \rightarrow \varepsilon$ (or arcs of the form *(e,])*). The remaining productions of the form $c(a) \rightarrow b\ c(b)$ (or arcs of the form *(a, b)*) shows the follows relation between the 1-sequences.

Example 4.3 (A 2-Reg). The following 6-tuple is a 2-Reg model for Example 2.1.

- $E = \{c1, x1, p1, p2\}$.
- $B = \{c, x, p\}$ where $c = d(c1)$, $x = d(x1)$ and $p = d(p1) = d(p2)$.
- $K = \{c1\ c1, c1\ x1, c1\ p1, x1\ c1, x1\ x1, x1\ p2, p1\ c1, p1\ x1, p1\ p1, p2\ c1, p2\ x1\}$.
- $C = \{S, c(c1\ c1), c(c1\ x1), c(c1\ p1), c(x1\ c1), c(x1\ x1), c(x1\ p2), c(p1\ c1), c(p1\ x1), c(p1\ p1), c(p2\ c1), c(p2\ x1)\}$.
- *S* is the start context.
- *P* contains 41 productions (See Figure 2.4).

Thus, the 2-Reg above models the same system modeled by the 1-Reg in Example 4.2 by using the same set of events. However, the semantics of the productions is different. For example, consider production $c(x1\ p2) \rightarrow p2\ x1\ c(p2\ x1)$ in Figure 2.4:

- *p2 x1* follows *x1 p2* in the 2-Reg.
- *x1* follows *x1 p2* in the system modeled by the 2-Reg.
- However, *p2 x1* does not follow *x1 p2* in the system modeled by the 2-Reg, because *p2* does not follow *p2* in the system, which can also be seen in Figure 2.2.

The event sequences generated using a k-Reg are contexted, whereas the sequences generated using an RG are not, because an RG generally uses basis events as terminals. Thus, decontexting function *d(.)* given in Definition 4.3 is extended from k-sequences to event sequences and to set of event sequences, in order to define and discuss the equivalence of k-Regs and RGs properly by associating (contexted) event sequences with basis event sequences.

Definition 4.4 (Decontexting Function). Given a k-Reg $G = (E, B, K, C, S, P)$. Let $s = s_1\ s_2 \ldots \in E^*$ be an *event sequence* and *X* be a set of event sequences. The corresponding *basis event sequence of s* is defined as

$$d(s) = d(s_1)\ d(s_2) \ldots \in B^* \text{ if } s \neq \varepsilon, \text{ and } d(\varepsilon) = \varepsilon.$$

The corresponding *set of basis event sequences of X* is defined as

$$d(X) = \{d(s) | \ s \in X\}.$$

Example 4.4 (Decontexted Event Sequences). Consider the 1-Reg in Figure 2.2.

- For event sequence $s = c1\ x1\ p2\ c1\ p1\ p1$, $d(s) = c\ x\ p\ c\ p\ p$.
- For set of event sequences $X = \{c1,\ c1\ p1,\ c1\ x1\ p2\}$, $d(X) = \{c,\ c\ p,\ c\ x\ p\}$.

Theorem 4.1 shows that the use of k-Regs does not cause any loss of generality with respect to RGs; that is, k-Regs and RGs are equivalent in the sense that they can both be used to represent same set of (noncontexted or basis) strings.

Theorem 4.1 (Equivalence of k-Regs and RGs). RGs and k-Regs are equivalent to each other.

Proof: To prove this, one needs to show that for each k-Reg G, there is an RG G' such that $d(L(G)) = L(G')$, and vice versa.

(From k-Reg to RG) Let $G = (E, B, K, C, S, P)$ be a k-Reg. To demonstrate how to convert a given k-Reg to an RG such that $d(L(G)) = L(G')$, the productions in P can be rewritten as follows:

$$P' = \{Q \rightarrow d(r)\ c(r) | \ Q \rightarrow r\ c(r) \in P\} \cup \{Q \rightarrow \varepsilon | \ Q \rightarrow \varepsilon \in P\}.$$

This replaces each contexted event by its corresponding basis event, because terminals of an RG are noncontexted events. Consequently, $G' = (C, E, P', S)$ is an extended right RG (see Definition 4.2) where $L(G') = d(L(G))$.

(From RG to 1-Reg) Without loss of generality, assume that $G = (N, E, P, S)$ is a right RG (see Definition 4.2). E corresponds to the set of basis events. Furthermore, the basis events and the nonterminals in the productions in P, and the empty string symbol ε are utilized to build a set of contexted events corresponding to these basis events. Each basis event in a production is paired up with the nonterminal that appears after it in this production, and with ε if no nonterminal appears after it. Thus, the set of contexted events (and also the set of 1-sequences) is given by

$$E' = K = \{e | \ Q \rightarrow x\ R \in P \text{ and } e = (x, R)\} \cup \{e | \ Q \rightarrow x \in P \text{ and } e = (x, \varepsilon)\}.$$

A unique context is assigned to each event in E' and the start context is included to obtain

$$C = \{c(e) | \ e \in E'\} \cup \{S\}.$$

Later, the set of productions is constructed by considering all the productions of the right RG as

$$P' = \{c(a) \rightarrow e\ c(e) | \ Q \rightarrow x\ R \in P,\ T \rightarrow y\ Q \in P,\ e = (x, R) \text{ and } a = (y, Q)\} \cup$$

$$\{c(a) \rightarrow e\ c(e),\ c(e) \rightarrow \varepsilon | \ Q \rightarrow x \in P,\ T \rightarrow y\ Q \in P,\ e = (x, \varepsilon) \text{ and } a = (y, Q)\} \cup$$

$$\{c(a) \rightarrow \varepsilon | \; Q \rightarrow \varepsilon \in P, T \rightarrow y\, Q \in P \text{ and } a = (y, Q)\} \cup$$
$$\{S \rightarrow e\, c(e) | \; S \rightarrow x\, R \in P \text{ and } e = (x, R)\} \cup$$
$$\{S \rightarrow e\, c(e), c(e) \rightarrow \varepsilon | \; S \rightarrow x \in P \text{ and } e = (x, \varepsilon)\} \cup$$
$$\{S \rightarrow \varepsilon | \; S \rightarrow \varepsilon \in P\}.$$

In this way, it is guaranteed that each (basis) sequence derived using productions in P can also be derived using productions in P' and applying decontexting function $d(.)$. Consequently, $G' = (E', E, K, C, S, P')$ is a 1-Reg (Definition 4.3) where $d(L(G)) = L(G')$.■

Since k-Regs are designed for event-based modeling and testing, event sequences that can and cannot be derived using k-Reg productions are distinguished for testing.

Definition 4.5 (Event Sequences in a k-Reg). Given a k-Reg $G = (E, B, K, C, S, P)$. Event sequence s is *in grammar* G, if there is a derivation of the form $Q \Rightarrow^* xsy$ for some $Q \in C$ and $x,y \in (C \cup E)^*$. A nonempty event sequence s in G is a *start sequence*, if there is a derivation of the form $S \Rightarrow^* s\, Q$ $(Q \in C)$, and it is a *finish sequence* if there is a derivation o the form $Q \Rightarrow^* s$ $(Q \in C)$. A start (or finish) 1-sequence is also a *start* (or *finish*) *event*.

An event sequence in G can be used to exercise some desirable or correct behavior, that is, a test path that is supposed to exist in the test object. An event sequence which is not in G and which can be used to exercise some undesirable or faulty behavior is also called a *faulty event sequence*. Thus, a faulty event sequence can be used to exercise a test path that is not supposed to exist in the test object.

Example 4.5 (Event Sequences in a 1-Reg). For the 1-Reg in Figure 2.2,

- set of some 2-sequences in the grammar is *{c1 x1, c1 p1, x1 p2, p1 p1}*.
- set of some 2-sequences not in the grammar is *{c1 p2, x1 p1, p2 p2}*.
- set of some start sequences is *{c1, x1, c1 c1, x1 x1 p2, c1 p1 c1, x1 p2 c1 p1}*.
- set of some finish sequences is *{p1, p2, p1 p1, c1 p1, x1 p2, c1 c1 x1 c1 p1}*.
- set of start events and set of finish events are *{c1, x1}* and *{p1, p2}*, respectively.

By Definition 4.3, given a k-Reg $G = (E, B, K, C, S, P)$, one can use

- a production of the form $Q \rightarrow r\, c(r)$ to obtain a k-sequence $r = r_1 \ldots r_k$.

- two productions the form $Q \rightarrow q\ c(q)$ and $c(q) \rightarrow r\ c(r)$ to obtain a (2k)-sequence $q\ r = q_1 \ldots q_k\ r_1 \ldots r_k$.

In general, using a derivation of length $m \geq 1$, one can obtain a (k×m)-sequence s such that $s \in K^*$ and s is in G. Each sequence (of arbitrary length) in G appears in one of such sequences. These sequences are important, for example, for test generation purposes. Definition 4.6 formally introduces them.

Definition 4.6 (m-derived Sequence). Given a k-Reg $G = (E, B, K, C, S, P)$. A (k×m)-sequence s such that $s \in K^*$ and s in G is called an *m-derived sequence* in grammar G, and integer $m \geq 1$ is the length of a derivation which yields s.

Example 4.6 (1-derived, 2-derived and 3-derived Sequences in a 2-Reg). For the 2-Reg in Figure 2.4

- $c1\ c1$ is a 1-derived sequence which can be obtained using production $c(c1\ c1) \rightarrow c1\ c1\ c(c1\ c1)$.
- $x1\ x1\ x1\ p2$ is a 2-derived sequence which can be obtained using productions $c(x1\ x1) \rightarrow x1\ x1\ c(x1\ x1)$ and $c(x1\ x1) \rightarrow x1\ p2\ c(x1\ p2)$ (in the given order).
- $p1\ x1\ x1\ p2\ p2\ c1$ is a 3-derived sequence which can be obtained using productions $c(c1\ p1) \rightarrow p1\ x1\ c(p1\ x1)$, $c(p1\ x1) \rightarrow x1\ p2\ c(x1\ p2)$ and $c(x1\ p2) \rightarrow p2\ c1\ c(p2\ c1)$ (in the given order).

Lemma 4.1 and Lemma 4.2 outline some basic properties of m-derived sequences in a given k-Reg. These lemmas are used to prove certain results in the following discussion.

Lemma 4.1 ("k-blocks" Property of an m-derived Sequence). Given a k-Reg $G = (E, B, K, C, S, P)$. If s is an m-derived sequence in G, then

$$s = u^1 \ldots u^m$$

where $m \geq 1$ and $u^i \in K$ for $i = 1, \ldots, m$, and $|s| = k \times m$.

Proof: Since s is an m-derived sequence in G, s can be obtained using a derivation of length $m \geq 1$ using the following productions

$$Q \rightarrow u^1\ c(u^1),\ c(u^1) \rightarrow u^2\ c(u^2),\ \ldots,\ c(u^{m-1}) \rightarrow u^m\ c(u^m)$$

in P in the given order (by Definition 4.6). Consequently,

$$Q \Rightarrow^* s\ c(u^m) \text{ with } s = u^1 \ldots u^m$$

where $m \geq 1$ and $u^i \in K$ for $i = 1, \ldots, m$ (by Definition 4.3), and $|s| = k \times m$.■

Lemma 4.2 ("Repetition" Property of an m-derived Sequence). Given a k-Reg $G = (E, B, K, C, S, P)$ and an m-derived sequence in G, $s = u^1 \ldots u^m$ where $u^i = u^i_1 \ldots u^i_k$ for $i = 1,\ldots, m$. If $k \geq 2$ and $m \geq 2$,

$$u^i_2 \ldots u^i_k = u^{i+1}_1 \ldots u^{i+1}_{k-1}$$

for $i = 1, \ldots, m-1$.

Proof: Since s is an m-derived sequence in G, $s = u^1 \ldots u^m$ where $m \geq 1$ and $u^i \in K$ for $i = 1, \ldots, m$ (by Lemma 4.1). It can be obtained using the following productions

$$Q \rightarrow u^1\ c(u^1),\ c(u^1) \rightarrow u^2\ c(u^2),\ \ldots,\ c(u^{m-1}) \rightarrow u^m\ c(u^m)$$

in P in the given order, where $u^i = u^i_1 \ldots u^i_k$ for $i = 1, \ldots, m$ (by Definition 4.6). If $k \geq 2$ and $m \geq 2$, for each production $c(u^i) \rightarrow u^{i+1}\ c(u^{i+1})$,

$$u^i_2 \ldots u^i_k = u^{i+1}_1 \ldots u^{i+1}_{k-1}$$

for $i = 1, \ldots, m-1$ (by Definition 4.3).■

Example 4.7 (Properties of a 3-derived Sequence in a 2-Reg). *s = p1 x1 x1 p2 p2 c1* can be derived using productions *c(c1 p1) → p1 x1 c(p1 x1)*, *c(p1 x1) → x1 p2 c(x1 p2)* and *c(x1 p2) → p2 c1 c(p2 c1)* in the 2-Reg in Figure 2.4 (in the given order). Hence, *s* is a 3-derived sequence in a 2-Reg where $|s| = 3 \times 2 = 6$. Also, u^1 = *p1 x2*, u^2 = *x1 p2* and u^3 = *p2 c1*, and $u^1_2 = u^2_1$ = *x1* and $u^2_2 = u^3_1$ = *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. This is carried out by exercising test paths which are or are not supposed to be in the test object using positive and negative test cases, respectively.

Definition 4.7 (Positive and Negative Test Cases). Given a k-Reg $G = (E, B, K, C, S, P)$.

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

- A *faulty complete event sequence (FCES)* is a negative test case which either is composed of only a nonstart event, or contains only a single 2-sequence which is not in G and it ends with this 2-sequence. $T_{FCES}(G) \subseteq T_N(G)$ denotes the *set of all FCESs*.
- A set of test cases is also called a *test set*.

Example 4.8 (Test Cases of a 1-Reg). For the 1-Reg in Figure 2.2,

- set of some positive test cases is *{x1, x1 x1, c1 p1, c1 p1 p1 x1}*.
- set of some CESs is *{x1 p2, x1 x1 p2, c1 p1, c1 p1 p1}*.
- set of some negative test cases is *{p1, c1 x1 p2 p2 c1, x1 c1 p1 c1 p2 p1}*.
- set of some FCESs is *{x1 p2 p2, x1 p2 p1, c1 x1 p2 p2}*.

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 (Definition 4.3). Thus, the equivalence of two k-Regs is defined by making use of decontexting function (Definition 4.4) and the set of all CESs (Definition 4.7) as follows.

Definition 4.8 (Equivalence). Two k-Regs G and H are equivalent, if $d(T_{CES}(G)) = d(T_{CES}(H))$.

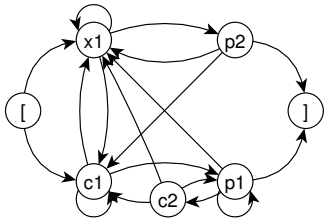
Example 4.9 (Equivalence of Two 1-Regs). The following is another 1-Reg model for Example 2.1, where contexted events corresponding to basis event c are obtained differently.

- $E = \{c1, c2, x1, p1, p2\}$.
- $B = \{c, x, p\}$.
- $K = E$, because $k = 1$.
- $C = \{S, c(c1), c(c2), c(x1), c(p1), c(p2)\}$.
- S is the start context.
- P contains *18* productions (See Figure 4.2).

Let G be the 1-Reg in Figure 2.2 and H be the 1-Reg in Figure 4.2. G and H are equivalent. The only difference is that, in H, c is distinguished in two ways, *c1* and *c2* by assigning two different contexts to c depending on whether it is performed after *p1* or not. Clearly, $T_{CES}(G) \neq T_{CES}(H)$, because G and H are not identical (*c2* is not an event in G but only in H). For this reason, one needs to use $d(T_{CES}(G)) = d(T_{CES}(H))$ to discuss their equivalence.

$S \rightarrow c1\ c(c1) \mid S \rightarrow x1\ c(x1)$
$c(c1) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid p1\ c(p1)$
$c(c2) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid p1\ c(p1)$
$c(x1) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid p2\ c(p2)$
$c(p1) \rightarrow c2\ c(c2) \mid x1\ c(x1) \mid p1\ c(p1) \mid \varepsilon$
$c(p2) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid \varepsilon$

(c) 1-Reg.



(d) Directed graph visualization.

Figure 4.2. Another 1-Reg model for Example 2.1.

Theorem 4.2 shows that k-Reg equivalence (Definition 4.8) is compatible to RG equivalence.

Theorem 4.2 (Compatibility of k-Reg Equivalence to RG Equivalence). Given two k-Regs G and H, and two RGs G' and H' which are equivalent to G and H, respectively. G and H are equivalent if and only if G' and H' are equivalent.

Proof: Since G is equivalent to G' and H is equivalent to H',

$$d(L(G)) = L(G') \text{ and } d(L(H)) = L(H')$$

(by Theorem 4.1). Also, $T_{CES}(G) = L(G)$ (by Definition 4.7).

(Only if part) If G and H are equivalent,

$$(d(T_{CES}(G)) = d(T_{CES}(H))) = (d(L(G)) = d(L(H))) = (L(G') = L(H')).$$

This shows that G' and H' are equivalent.

(If part) If G' and H' are equivalent,

$$(L(G') = L(H')) = (d(L(G)) = d(L(H))) = (d(T_{CES}(G)) = d(T_{CES}(H))),$$

which shows that G and H are equivalent.■

In practice, it is important to make sure that a model and its elements are utilized completely. Otherwise, some elements may turn out to be irrelevant and they may not be used in testing process; correct functioning of the test object cannot be checked for certain behaviors because certain test paths are excluded. For this reason, usefulness is defined as follows.

Definition 4.9 (Usefulness). Given a k-Reg $G = (E, B, K, C, S, P)$. A string $z \in (C \cup E)^*$ is *useful in grammar* G, if $S \Rightarrow^* xzy \Rightarrow^* w$ for some $x,y \in (C \cup E)^*$ and $w \in E^*$. G is *useful*, if all k-sequences in K are useful in G.

Example 4.10 (Useful and Nonuseful k-Regs). k-Regs given in Figure 2.2, Figure 2.4 and Figure 4.2 are all useful. To obtain a nonuseful 1-Reg from Figure 2.2, one can remove prodcutions $c(p1) \rightarrow \varepsilon$ and $c(p2) \rightarrow \varepsilon$ from the grammar. The resulting grammar does not have any finish events anymore. Therefore, $T_{CES}(G)$ is empty, but the follows relation between events is still described correctly.

As demonstrated in Example 4.10, in the absence of finish k-sequences in a k-Reg, the set of all positive test cases does not change. However, the set of all CESs becomes empty, because all k-sequences lose usefulness. For this reason, the presence of finish k-sequences is important to be able to generate CESs. Although it depends on the system, one can even select the finish events arbitrarily to obtain a useful model.

Due to the form of a k-Reg (Definition 4.3), by ensuring the usefulness of each k-sequence, it is guaranteed that all the contexts and the productions can be utilized. Thus, each event sequence in G can be contained in a test case in $T_{CES}(G)$. Theorem 4.3 demonstrates this result.

Theorem 4.3 (Sequences in a Useful k-Reg). Given a k-Reg $G = (E, B, K, C, S, P)$. If G is useful, then each nonempty sequence s in G is contained in a CES in G.

Proof: If all k-sequences in K are useful, then all nonempty sequences in G are useful. Thus, given a nonempty sequence s is in G, $S \Rightarrow^* xsy \Rightarrow^* w$ for some $x,y \in (C \cup E)^*$ and $w \in E^*$. Thus, s is contained in a CES in G.■

Theorem 4.3 implies that, if G is a useful k-Reg, each nonempty positive test G can be contained in a CES in G. Hence, the following is a corollary to Theorem 4.3.

Corollary 4.1 (Positive Test Cases of a Useful k-Reg). Given a k-Reg $G = (E, B, K, C, S, P)$. If G is useful, then each nonempty test case in $T_P(G)$ is contained in a test case in $T_{CES}(G)$.

Proof: The proof follows from Theorem 4.3, because each nonempty positive test case is a nonempty sequence in G.■

Using Corollary 4.1, one can make use of the generative nature of grammars to obtain CESs and use them for testing purposes.

Another property of the system model that is considered as helpful is determinism. Deterministic system models help to exclude redundant event sequences from the model. Thus, testing process becomes more efficient, because multiple test cases that exercise the same test paths in the test object are excluded. A deterministic k-Reg is defined as follows.

Definition 4.10 (Determinism). A k-Reg $G = (E, B, K, C, S, P)$ 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 $d(r) = d(q)$.

By using a deterministic k-Reg G, one can guarantee that each m-derived start sequence in the grammar has a unique basis sequence. Theorem 4.4 proves this statement.

Theorem 4.4 (m-derived Start Sequences in a Deterministic k-Reg). Given a k-Reg $G = (E, B, K, C, S, P)$. If G is deterministic, there exist no two m-derived start sequence s and t such that $s \neq t$ and $d(s) = d(t)$.

Proof: This can be proved by contradiction. Assume that G is deterministic, and let s and t be two m-derived start sequences in G such that $s \neq t$ and $d(s) = d(t)$. s and t can be written as

$$s = u^1 \ldots u^m$$

where $u^i \in K$ for $i = 1, \ldots, m$, and

$$t = v^1 \ldots v^m$$

where $v^i \in K$ for $i = 1, \ldots, m$ (by Lemma 4.1). Since $s \neq t$ and $d(s) = d(t)$, let j be the smallest index such that

$$u^j \neq v^j \text{ and } d(u^j) = d(v^j).$$

Let $Q = S$, if $j = 1$; $Q = c(u^{j-1}) = c(v^{j-1})$, otherwise. Since $u^i = v^i$ for $i = 1, \ldots, j-1$,

$$Q \rightarrow u^j\ c(u^j) \in P \text{ and } Q \rightarrow v^j\ c(v^j) \in P$$

where $u^j \neq v^j$ and $d(u^j) = d(v^j)$. This contradicts the fact that G is deterministic.■

As an implication of Theorem 4.4, the use of redundant positive test cases or CESs is avoided by utilization of deterministic models. Thus, the following is a corollary to Theorem 4.4.

Corollary 4.2 (m-derived Positive Test Cases of a Deterministic k-Reg). Given a k-Reg $G = (E, B, K, C, S, P)$. If G is deterministic, each m-derived positive test case in $T_P(G)$ and each nonempty test case in $T_{CES}(G)$ has a unique basis.

Proof: The proof follows from Theorem 4.4, because each m-derived positive test case in G is an m-derived start sequence in G, and so is each CES in G.■

Corollary 4.2 suggests that, a benefit of using a deterministic k-Reg G is to have a unique basis $d(s)$ for each positive test case s such that s is an m-derived sequence in G. Another benefit is to avoid generation of *masked negative test*

cases which are negative test cases that do not exercise any faulty behavior; that is, they exercise test paths which are supposed to exist in the test object like positive test cases. Example 4.11 demonstrates these two benefits.

Example 4.11 (Test Cases of Deterministic and Nondeterministic 1-Regs). Let G be the deterministic 1-Reg in Figure 2.2 and H be the nondeterministic 1-Reg which is obtained by including production $c(p1) \rightarrow p2\ c(p2)$ in G. Furthermore, let $s = c1\ p1\ p1$ and $t = c1\ p1\ p2$ two test cases.

- With respect to H: s and t are positive test cases. However, they are redundant, because $d(s) = d(t) = c\ p\ p$.
- With respect to G: s is a positive test case and t is a negative test case. However, t does not exercise a faulty behavior because there is a positive test case s in G such that $d(s) = d(t)$. Therefore, t is actually a masked negative test case.

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

5 Varying Morphology

Based on Definition 4.3, a (k+1)-Reg model is morphologically different from a k-Reg model, and it can be used to model or reveal different or more subtle faults, that is, faults that are located on different test paths in the test object (as discussed in Section 2.4). For this purpose, this chapter outlines a transformation to vary k and generate models with morphological differences and also discusses coverage-based test generation from these models.

5.1 Grammar Transformation to Vary Morphology

In this section, a transformation of a given k-Reg to its corresponding (k+1)-Reg is given; this transformation can also be used to transform a given 1-Reg model to its corresponding k-Reg model. Furthermore, relations between m-derived sequences in morphologically different models that describe the same system are studied; these relations are important mainly for test generation.

Before giving the definition of k-Reg transformation, the following observations are made using the 1-Reg in Figure 2.2 and its corresponding 2-Reg in Figure 2.4.

- For each production of the form $c(q_1\ q_2) \rightarrow r_1\ r_2\ c(r_1\ r_2)$ in Figure 2.4, $q_2 = r_1$ (by Definition 4.3). Thus, each such production can be obtained by combining two productions $c(q_1) \rightarrow q_2\ c(q_2)$ and $c(q_2) \rightarrow r_2\ c(r_2)$ in Figure 2.2.
- Each production of the form $S \rightarrow r_1\ r_2\ c(r_1\ r_2)$ in Figure 2.4 can be obtained by combining two productions $S \rightarrow r_1\ c(r_1)$ and $c(r_1) \rightarrow r_2\ c(r_2)$ in Figure 2.2.
- Each production of the form $c(r_1\ r_2) \rightarrow \varepsilon$ in Figure 2.4 can be obtained by combining two productions $c(r_1) \rightarrow r_2\ c(r_2)$ and $c(r_2) \rightarrow \varepsilon$ in Figure 2.2.

k-Reg transformation is given in Definition 5.1 by generalizing these observations.

Definition 5.1 (k-Reg Transformation). Given a 1-Reg $G_1 = (E, B, K_1, C_1, S, P_1)$.

- *The corresponding 1-Reg of* G_1 is defined as itself:

  $$G_1 = (E, B, K_1, C_1, S, P_1).$$

- Let $G_k = (E, B, K_k, C_k, S, P_k)$ be *the corresponding k-Reg of* G_1. *The corresponding (k+1)-Reg of* G_1 *(or* G_k*)* is defined as

  $$G_{k+1} = (E, B, K_{k+1}, C_{k+1}, S, P_{k+1}) \text{ where}$$

  - $K_{k+1} = \{q_1 \dots q_k r_k | \; c(q) \rightarrow r\, c(r) \in P_k$ where $q = q_1 \dots q_k$ and $r = r_1 \dots r_k\}$ is the set of all (k+1)-sequences in G_1.
  - $C_{k+1} = \{c(r) | \; r \in K_{k+1}\}$ is the set of contexts.
  - $P_{k+1} = \{S \rightarrow r\, e\, c(r\, e) | \; S \rightarrow r\, c(r) \in P_k$ and $c(r_k) \rightarrow e\, c(e) \in P_1\} \cup \{c(q\ r_k) \rightarrow \varepsilon | \; c(q) \rightarrow r\ c(r) \in P_k$ and $c(r_k) \rightarrow \varepsilon \in P_1\} \cup \{c(q\, r_k) \rightarrow r\, e\, c(r\, e) | \; c(q) \rightarrow r\, c(r) \in P_k$ and $c(r_k) \rightarrow e\, c(e) \in P_1\}$ is the set of productions.

Based on Definition 5.1, Algorithm 5.1 outlines the steps to perform k-Reg transformation.

Algorithm 5.1. k-Reg Transformation

Input: $G_k = (E, B, K_k, C_k, S, P_k)$ – the input k-Reg (the corresponding k-Reg of G_1)
$G_1 = (E, B, K_1, C_1, S, P_1)$ – the input 1-Reg
Output: $G_{k+1} = (E, B, K_{k+1}, C_{k+1}, S, P_{k+1})$ – the corresponding (k+1)-Reg
$K_{k+1} = \varnothing$, $C_{k+1} = \{S\}$, $P_{k+1} = \varnothing$
for each $Q \rightarrow r\, c(r) \in P_k$ where $r = r_1 \dots r_k$ **do**
  if $Q = c(q)$ where $q = q_1 \dots q_k$ **then**
    $K_{k+1} = K_{k+1} \cup \{q\, r_k\}$
    $C_{k+1} = C_{k+1} \cup \{c(q\, r_k)\}$
  endif
  for each $c(r_k) \rightarrow R \in P_1$ **do**
    if $R = e\, c(e)$ **then**
      if $Q = S$ **then**
        $P_{k+1} = P_{k+1} \cup \{S \rightarrow r\, e\, c(r\, e)\}$
      else if $Q = c(q)$ **then**
        $P_{k+1} = P_{k+1} \cup \{c(q\, r_k) \rightarrow r\, e\, c(r\, e)\}$
      endif
    else if $R = \varepsilon$ **then**
        $P_{k+1} = P_{k+1} \cup \{c(q\, r_k) \rightarrow \varepsilon\}$
    endif
  endfor
endfor

Algorithm 5.1 runs in $O(k\ |P_1|\ |P_k|) = O(k\ |P_1|\ |P_1|^k) = O(k\ |P_1|^{k+1})$ worst case time due to the following.

- $|P_k| = O(|P_1|^k)$.
- All set union operations can be performed in $O(1)$ time as append operations, because a different element is added to the each set during each union operation.
- Construction of a (k+1)-sequence in G_1 can be performed in $O(k)$ steps by merging a k-sequence in G_1 extracted from G_k with a 1-sequence in G_1 extracted from G_1.

This time complexity is expected, because the number of k-sequences increases exponentially in k. In practice, however, $|P_k|$ is generally much smaller than $|P_1|^k$ and k is almost always bounded. Hence, transformation can be performed quite fast for properly selected k values.

Also, transformation of a k-Reg to its corresponding (k+1)-Reg can be performed by only using the given k-Reg, by extracting both k-sequences and 1-sequences from the productions of this k-Reg. However, this causes the complexity to increase to $O(k\ |P_k|\ |P_k|) = O(k\ |P_1|^{2k})$. Therefore, the details are skipped.

Example 5.1 (A Corresponding 2-Reg). The 2-Reg in Figure 2.4 is the corresponding 2-Reg of the 1-Reg in Figure 2.2.

As Definition 5.1 suggests (Also see Algorithm 5.1):

- A new (k+1)-sequence $q_1 \dots q_k\ r_k$ in G_1 is extracted from $c(q_1 \dots q_k) \rightarrow r_1 \dots r_k\ c(r_1 \dots r_k) \in P_k$ using the fact that $q_1 \dots q_k$ is a k-sequence in G_1 and $q_k\ r_k$ is a 2-sequence in G_1 (that is, $c(q_k) \rightarrow r_k\ c(r_k) \in P_1$). In this way, all (k+1)-sequences in G_1 are obtained.
- To determine the contexts to be used in a new production properly, a production from G_k and a production from G_1 are selected and used in such a way that (k+1)-sequences that are not in G_1 does not emerge, and all (k+1)-sequences in G_1 are included in new productions together with their contexts without invalidating the definition of a k-Reg.
- k-sequences in G_1 which cannot be included in some (k+1)-sequences in G_1 are left out; that is, $r \in K_k$ is excluded if and only if $S \rightarrow r\ c(r) \in P_k$ and $c(r) \rightarrow \varepsilon \in P_k$ is the only production of the form "$c(r) \rightarrow \dots$" in P_k is (implying that $c(r_k) \rightarrow \varepsilon \in P_1$).
- $S \rightarrow \varepsilon$ is not included in P_{k+1}.

There are also some special cases which follow from Definition 5.1. Let $G_1 = (E, B, K_1, C_1, S, P_1)$ be a 1-Reg, $G_k = (E, B, K_k, C_k, S, P_k)$ be its corresponding k-Reg and $G_{k+1} = (E, B, K_{k+1}, C_{k+1}, S, P_{k+1})$ be its corresponding (k+1)-Reg.

- $P_k = \{S \rightarrow q\ c(q),\ c(q) \rightarrow \varepsilon\}$ implies that there is no production of the form $c(q_k) \rightarrow e\ c(e) \in P_1$; otherwise, there would be other productions using k-sequence $q_2 \ldots q_k\ e$ in P_k. Consequently, $K_{k+1} = \varnothing$ and $P_{k+1} = \varnothing$.
- Similarly, $P_k = \{S \rightarrow q\ c(q),\ c(q) \rightarrow r\ c(r),\ c(r) \rightarrow \varepsilon\}$ implies that $K_{k+1} = \{q\ r_k\}$ and $P_{k+1} = \{S \rightarrow q\ r_k\ c(q\ r_k),\ c(q\ r_k) \rightarrow \varepsilon\}$.

Some important properties of m-derived sequences in a corresponding k-Reg of a given 1-Reg are discussed in Lemma 5.1.

Lemma 5.1 (m-derived Sequences in a Corresponding k-Reg). Given a 1-Reg $G_1 = (E, B, K_1, C_1, S, P_1)$ and its corresponding k-Reg $G_k = (E, B, K_k, C_k, S, P_k)$ where $k \geq 2$ and $K_k \neq \varnothing$. If $s = u^1 \ldots u^m$ is an m-derived sequence where $u^i = u^i_1 \ldots u^i_k \in K_k$ for $i = 1, \ldots, m$, then

$u^i_j\ u^i_{j+1}$ is a 2-sequence in G_1 for $j = 1, \ldots, k\text{-}1$ and $i = 1, \ldots, m$, and,

if $m \geq 2$, $u^i_k\ u^{i+1}_k$ is a 2-sequence in G_1 for $i = 1, \ldots, m\text{-}1$.

Proof: Since $s = u^1 \ldots u^m$ is an m-derived sequence,

$$u^i \in K_k,$$

for $i = 1, \ldots, m$ (by Definition 5.1); that is, each u^i is a k-sequence in G_1. Therefore, each $u^i_j\ u^i_{j+1}$, that is, each 2-sequence in u^i, is a 2-sequence in G_1 for $j = 1, \ldots, k\text{-}1$ and $i = 1, \ldots, m$.

In addition, if $m \geq 2$,

$$c(u^i) \rightarrow u^{i+1}\ c(u^{i+1}) \in P_k$$

for $i = 1, \ldots, m\text{-}1$ (by Definition 4.6). Consequently,

$$c(u^i_k) \rightarrow u^{i+1}_k\ c(u^{i+1}_k) \in P_1$$

for $i = 1, \ldots, m\text{-}1$ (by Definition 5.1). Hence, $u^i_k\ u^{i+1}_k$ is a 2-sequence in G_1 for $i = 1, \ldots, m\text{-}1$.■

Example 5.2 (An m-derived Sequence in a Corresponding 2-Reg). *s = p1 x1 x1 p2 p2 c1* is in the 2-Reg given in Figure 2.4. Thus, 2-sequences *p1 x1*, *x1 p2* and *p2 c1* are in the 1-Reg given in Figure 2.2.

Note that *s* in Example 5.2 is not in the 1-Reg given in Figure 2.2, because *p2 p2* is not in this 1-Reg. Nevertheless, one can extract a sequence in the 1-Reg *t = p1 x1 p2 c1* from *s* by deleting 1-sequence from the beginning of each 2-sequence in *s*, except for the first one. Thus, in general, a sequence which is in the corresponding k-Reg of a given 1-Reg need not be a sequence in this 1-Reg.

However, it is possible to transform a sequence in the k-reg and obtain a sequence in the 1-Reg. For this purpose, sequence transformations are defined as follows.

Definition 5.2 (Sequence Transformations). Given a (k+m-1)-sequence s where $k \geq 1$ and $m \geq 1$. *Sequence transformation of s based on integer k* is defined as a (k×m)-sequence

$$T_S(s, k) = u^1 \ldots u^m$$

where $u^i = u^i_1 \ldots u^i_k = s_i \ldots s_{i+k-1}$ for $i = 1, \ldots, m$.

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

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

where $u^1 = s_1 \ldots s_k$ and each $u^i_k = s_{i \times k}$ for $i = 2, \ldots, m$.

For a (k+m-1)-sequence s where $k \geq 1$ and $m \geq 1$, Algorithm 5.2 can be applied to transform s into $T_S(s, k)$. Computation of $T_S(s, k)$ can be performed in $O(k\,m)$ time.

Algorithm 5.2. $T_S(s, k)$ – Sequence Transformation

Input: s – the (k+m-1)-sequence where $k \geq 1$ and $m \geq 1$
$k \geq 1$ – the integer based on which transformation is performed
Output: t – the transformed (k×m)-sequence
t is a sequence of length $k \times m$
for $i=1$ **to** m
$d = (i-1) \times k$
$t_{d+1} \ldots t_{d+k} = s_i \ldots s_{i+k-1}$
endfor

Algorithm 5.3. $T_S^{-1}(s, k)$ **–** Inverse Sequence Transformation

Input: s – the (k×m)-sequence where $k \geq 1$ and $m \geq 1$
$k \geq 1$ – the integer based on which transformation is performed
Output: t – the transformed (k+m-1)-sequence
t is a sequence of length (k+m-1)
$t_1 \ldots t_k = s_1 \ldots s_k$
for $i=1$ **to** $m-1$
$t_{k+i} = s_{k \times (i+1)}$
endfor

In addition, Algorithm 5.3 shows the steps to perform inverse sequence transformation on a (k×m)-sequence s where $k \geq 1$ and $m \geq 1$. Computation of $T_S^{-1}(s, k)$ has the worst case time complexity of $O(k + m-1)) = O(k + m)$.

Theorem 5.1 establishes relations between (k+m-1)-derived sequences in a 1-Reg and m-derived sequences in its corresponding k-Reg. More precisely, an m-derived sequence in a corresponding k-Reg can be used to obtain to a (k+m-1)-derived sequences in its 1-Reg, and vice versa.

Theorem 5.1 (Relation between (k+m-1)-derived Sequences in 1-Reg and m-derived Sequences in Its Corresponding k-Reg). Given a 1-Reg $G_1 = (E, B, K_1, C_1, S, P_1)$ and it corresponding k-Reg $G_k = (E, B, K_k, C_k, S, P_k)$ where $k \geq 2$ and $K_k \neq \varnothing$. Let s and t be two sequences such that $t = T_S^{-1}(s, k)$ and $s = T_S(t, k)$. s is an m-derived sequence in G_k if and only if t is a (k+m-1)-derived sequence in G_1.

Proof: Only if part and if part can be proved separately as follows.

(Only if part) If s is an m-derived sequence in G_k,

$$s = u^1 \ldots u^m$$

where $m \geq 1$ and $u^i = u^i_1 \ldots u^i_k \in K_k$ for $i = 1, \ldots, m$ (by Lemma 4.1). Also,

$$t = T_S^{-1}(s, k) = u^1 u^2_k \ldots u^m_k.$$

(by Definition 5.2). Hence, only if part follows from the fact that

$$u^1 \text{ is a k-sequence in } G_1$$

(by Definition 5.1) and that

$$u^i_k u^{i+1}_k \text{ is a 2-sequence in } G_1$$

for $i = 1, \ldots, m-1$, when $m \geq 2$ (by Lemma 5.1).

(If part) If t is a (k+m-1)-derived sequence in G_1 where $k \geq 1$ and $m \geq 1$,

$$t = t_1 \ldots t_{k+m-1}$$

(by Lemma 4.1). Let

$$u^i = u^i_1 \ldots u^i_k = t_i \ldots t_{i+k-1}$$

for $i = 1, \ldots, m$, then

$$s = T_S(t, k) = u^1 \ldots u^m.$$

(by Definition 5.2). Thus, if part follows from the fact that

$$u^i \in K_k$$

for $i = 1, \ldots, m$ and that

$$u^i_2 \ldots u^i_k = u^{i+1}_1 \ldots u^{i+1}_{k-1}$$

for $i = 1, \ldots, m-1$ when $m \geq 2$.■

Example 5.3 (Sequence Transformations). Let G be the 1-Reg in Figure 2.2 and H be its corresponding 2-Reg in Figure 2.4. By Theorem 5.1:

- For a 4-derived sequence $s = c1\ c1\ c1\ x1\ x1\ x1\ x1\ p2$ in H, $T_S^{-1}(s, 2) = c1\ c1\ x1\ x1\ p2$ is a 5-derived sequence in G.
- For a 3-derived sequence $s = x1\ p2\ x1$ in G, $T_S(s, 2) = x1\ p2\ p2\ x1$ is a 2-derived sequence in H.

The following corollary to Theorem 5.1 can be used to to employ this result in test generation.

Corollary 5.1 (Relation between (k+m-1)-derived Positive Test Cases in a 1-Reg and m-derived Positive Test Cases in Its Corresponding k-Reg). Given a 1-Reg $G_1 = (E, B, K_1, C_1, S, P_1)$ and it corresponding k-Reg $G_k = (E, B, K_k, C_k, S, P_k)$ where $k \geq 2$ and $K_k \neq \varnothing$. Let s and t be two sequences such that $t = T_S^{-1}(s, k)$ and $s = T_S(t, k)$. s is an m-derived sequence in G_k if and only if t is a (k+m-1)-derived sequence in G_1. For each m-derived sequence $s \in T_P(G_k)$ (or $s \in T_{CES}(G_k)$), there is an (m+k-1)-derived sequence $t \in T_P(G_1)$ (or $t \in T_{CES}(G_1)$) such that $t = T_S^{-1}(s, k)$, and vice versa.

Proof: The proof follows from Theorem 5.1, since all start and finish events of G_k is a subset of all the start and finish events of G_1, respectively (by Definition 5.1).■

By Corollary 5.1, each test case which is an m-derived sequence in the corresponding k-Reg of a given 1-Reg can be used to build a test case for the 1-Reg.

As mentioned in Chapter 4, usefulness and determinism properties are important. The following theorem gives the sufficient conditions for usefulness and determinism of the corresponding k-Reg of a given 1-Reg, showing that k-Reg transformation (See Definition 5.1) preserves both of these properties.

Theorem 5.2 (Usefulness and Determinism of a Corresponding k-Reg). Given a 1-Reg $G_1 = (E, B, K_1, C_1, S, P_1)$ and its corresponding k-Reg $G_k = (E, B, K_k, C_k, S, P_k)$ where $k \geq 2$ and $K_k \neq \varnothing$.

1. If G_1 is useful, then G_k is also useful.
2. If G_1 is deterministic, then G_k is also deterministic.

Proof: Each case can be proved by contrapositive as follows.

1. If G_k is not useful, then there exists a k-sequence

   $$r = r_1 \ldots r_k \in K_k \text{ in } G_k$$

   such that r is not included in any CES in G_k (by Definition 4.9). In this case,

$$r = r_1 \ldots r_k = T_S^{-1}(r, k) \text{ in } G_1$$

is not included in any CES in G_1 either (by Corollary 5.1). Hence, at least the last event of r, that is, r_k, is not useful in G_1. Therefore, G_1 is not useful, which completes the proof of 1.

2. If G_k is not deterministic, then there exist two productions

$$Q \rightarrow q\ c(q) \in P_k \text{ and } Q \rightarrow r\ c(r) \in P_k$$

such that

$$r \neq q \text{ and } d(r) = d(q)$$

(by Definition 4.10). Now, let j be the smallest index such that

$$r_j \neq q_j \text{ and } d(r_j) = d(q_j).$$

In this case, since $r_j \in K_1$ and $q_j \in K_1$ are in G_1, there exists $R \in C_1$ such that

$$R \rightarrow r_j\ c(r_j) \in P_1 \text{ and } R \rightarrow q_j\ c(q_j) \in P_1$$

where

$$R = S \text{ if } j = 1 \text{ and } Q = S,$$
$$R = c(s_1) \text{ if } j=1 \text{ and } Q = c(s_1 \ldots s_k), \text{ and}$$
$$R = c(r_{j-1}) \text{ if } j > 1$$

(by Definition 5.1). Thus, G_1 is not deterministic, which completes the proof of 2.■

5.2 Test Generation from Morphologically Different Models

Morphologically different k-Reg models which are obtained by varying k can be used to generate positive test cases that intend to reveal different or more subtle faults. Positive test cases derived from a (k+1)-Reg exercise different test paths that are supposed to exist in the test object than those derived from a k-Reg. In the following, some coverage criteria are discussed for event-based testing (using k-Regs) to systematize the test process, to judge the efficiency of the test cases and to determine when to stop testing. Furthermore, related results and test generation methods are also given to demonstrate that the approach is sound.

Two immediate coverage criteria which are inherited from grammar-based testing are terminal coverage and production coverage.

Definition 5.3 (Terminal Coverage). Given a k-Reg $G = (E, B, K, C, S, P)$ and a set of sequences $X \subseteq T_P(G)$. X is said to *cover a terminal* $e \in K$, if e appears at

least in one of the sequences in X. If X covers all terminals in K, it is said to achieve *terminal coverage*.

Definition 5.4 (Production Coverage). Given a k-Reg $G = (E, B, K, C, S, P)$ and a set of sequences $X \subseteq T_P(G)$. X is said to *cover a production* $p \in P$, if p is used at least once in a derivation of a sequence in X. If X covers all productions in P, it is said to achieve *production coverage*.

For a given k-Reg, achieving production coverage is sufficient condition for covering all pairs of terminals, because productions form a "follows" relation between terminals (See Definition 4.3). Thus, production coverage subsumes terminal pair coverage, and, so, terminal coverage.

Note that Definition 5.3 uses the term "terminal" instead of "k-sequence" (See Definition 4.3), because the term "k-sequence coverage" is reserved for only 1-Regs. This point is not elaborated further by define coverage of terminal sequences of some fixed length because production coverage of a corresponding k-Reg of a given 1-Reg is used as sufficient condition to cover all (k+1)-sequences in the given 1-Reg (or in a system which is modeled by the 1-Reg). Therefore, the following is defined.

Definition 5.5 (k-sequence Coverage). Given a 1-Reg $G = (E, B, K, C, S, P)$ and a set of sequences $X \subseteq T_P(G)$. X is said to cover a k-sequence r in G, if r appears in a sequence in X. If X covers all k-sequences in G, it is said to achieve *k-sequence coverage*.

k-sequence coverage is used to reveal missing event faults where an event does not follow or precede a (possibly empty) sequence of events. It is based on contexted events; therefore, it subsumes basis k-sequence coverage. Although, k-sequence coverage can be used to reveal different or more subtle faults as k is increased, it is not stronger for increasing value of k; that is, (k+1)-sequence coverage does not subsume k-sequence coverage for $k \geq 1$. As discussed in Section 5.1, it is possible that a k-sequence is not included in any (k+1)-sequences. In this case, a sequence set achieving m-sequence coverage for $m \geq k+1$ fails to cover such k-sequences. If a complete subsumption is intended, such sequences should be singled out and included separately.

In order to generate test cases achieving (k+1)-sequence coverage from a given 1-Reg, its corresponding k-Reg can be used. Before giving the complete test generation algorithm, the sufficient and necessary conditions to check if (k+1)-sequence coverage is achievable for a given 1-Reg are outlined in Lemma 5.2.

Lemma 5.2 (Achievability of (k+1)-sequence Coverage). Given a 1-Reg $G_1 = (E, B, K_1, C_1, S, P_1)$ and its corresponding k-Reg $G_k = (E, B, K_k, C_k, S, P_k)$ where $k \geq 1$ and $K_k \neq \varnothing$. (k+1)-sequence coverage for G_1 is achievable if and only if there exists at least one m-derived sequence s in G_k such that $|s| = k\ m$ for $m \geq 2$.

Proof: If part and only if part can be proved separately as follows.

(If part) Let s be an m-derived sequence in G_k such that $|s| = k\ m$ for $m \geq 2$. Then $T_S^{-1}(s, k)$ is a (k+m-1)-sequence in G_1 (by Theorem 5.1), which implies that there exists a (k+1)-sequence in G_1, and thus (k+1)-sequence coverage for G_1 is achievable.

(Only if part) Let s be a (k+1)-sequence in G_1. Then $t = T_S(s, k)$ is a (2k)-sequence or 2-derived sequence in G_k (by Theorem 5.1).■

Unless noted otherwise, (k+1)-sequence coverage is assumed to be achievable for all 1-Regs under consideration. Below, sufficient and necessary conditions to achieve (k+1)-sequence coverage for a given 1-Reg are given.

Theorem 5.3 (Achieving (k+1)-sequence Coverage). Given a 1-Reg $G_1 = (E, B, K_1, C_1, S, P_1)$, its corresponding k-Reg $G_k = (E, B, K_k, C_k, S, P_k)$ where $k \geq 1$ and $K_k \neq \varnothing$, a set of sequences $X \subseteq T_P(G_1)$ and a set of transformed sequence $T_S(X, k) = \{T_S(s, k) | \ s \in X\} \subseteq T_P(G_k)$. X achieves (k+1)-sequence coverage for G_1 if and only if $T_S(X, k)$ achieves terminal pair coverage for G_k; that is, $T_S(X, k)$ covers all pairs of k-sequences rq in G_k such that $r,q \in K_k$.

Proof: If part and only if part can be proved separately as follows.

(If part) For each terminal pair pq in G_k, $T_S^{-1}(pq, k) = pq_k$ is a (k+1)-sequence in G_1 (by Theorem 5.1). Since $T_S(X, k)$ covers all terminal pairs in G_k, $T_S^{-1}(T_S(X, k), k) = \{T_S^{-1}(s, k) | \ s \in T_S(X, k)\} = X$ covers all (k+1)-sequences in G_1 (by Definition 5.1).

(Only if part) For each (k+1)-sequence s in G_1, $T_S(s, k)$ is a (2k)-sequence such that $s \in K_k^*$ (by Theorem 5.1); that is, s is a terminal pair. Since X achieves (k+1)-sequence coverage for G_1, $T_S(X, k)$ achieves terminal pair coverage for G_k (by Definition 5.1).■

Theorem 5.3 is a strong result. It entails coverage of only a certain subset of productions, because coverage of some productions may not lead to coverage of additional (k+1)-sequences. However, the selection of this specific subset of productions requires additional effort, and, thus, this result is relatively harder to use. For this reason, the following corollary to Theorem 5.3 can be used to construct an algorithm for generation of test cases achieving (k+1)-sequence coverage for a given 1-Reg.

Corollary 5.2 (Achieving (k+1)-sequence Coverage - Weaker). Given a 1-Reg $G_1 = (E, B, K_1, C_1, S, P_1)$, its corresponding k-Reg $G_k = (E, B, K_k, C_k, S, P_k)$ where $k \geq 1$ and $K_k \neq \varnothing$, a set of sequences $X \subseteq T_P(G_1)$ and a set of transformed sequence $T_S(X, k) = \{T_S(s, k) | s \in X\} \subseteq T_P(G_k)$. X achieves (k+1)-sequence coverage for G_1, if $T_S(X, k)$ achieves production coverage for G_k.

Proof: The proof follows from Theorem 5.3 and the fact that production coverage subsumes terminal pair coverage for a given k-Reg (See the discussion that follows Definition 5.4).■

Based on Corollary 5.2, Algorithm 5.4 can be used to generate a test set that achieves (k+1)-sequence coverage for the given 1-Reg. In this way, one can reveal missing event faults where an event does not follow or precede a certain k-sequence by using the generated test cases to exercise test paths which are supposed to exist in the test object and check whether the test object functions correctly. Note that Algorithm 5.4 uses Algorithm 5.1 to transform k-Reg, an external production-covering algorithm to generate a set of sequences, and Algorithm 5.3 to perform inverse sequence transformations on the generated sequences.

Algorithm 5.4. Test Generation to Achieve (k+1)-sequence Coverage

Input: $G = (E, B, K, C, S, P)$ – the input 1-Reg
k – an integer ≥ 1
Output: X – a set of sequences which achieves (k+1)-sequence coverage for G
$X = \varnothing$
G_k = transform G to its corresponding k-Reg //See Algorithm 5.1
Y = generate a sequence set achieving production coverage for G_k
for each $s \in Y$ such that $|s| \geq 2k$ **do**
$X = X \cup T_S^{-1}(s, k)$ //See Algorithm 5.3
endfor

The test cases generated by Algorithm 5.4 are CESs in the given grammar G. It is possible to perform some optimizations. For example, algorithms to solve Chinese Postman Problem over directed graphs, like [114][65][7], can be adapted to cover each production a minimum number of times, resulting in a reduced set of test cases. However, one should note that optimization algorithms tend to require more resources in terms of both time and space, and there is no guarantee of reduced test execution costs [32][33]. Thus, algorithms such as those in [146][119][176] can also be used to generate relatively short but generally nonoptimized sequences from a given grammar, while using less resources.

Due to the nature of (k+1)-sequence coverage, no matter which type of method is used to cover productions in the given grammar, performance of Algorithm 5.4 quickly declines with increasing k. The worst-case running time complexity is given by

$$O((k-1)|P|^{k-1} + C_P(|E|, |P|, k) + C_T(|E|, |P|, k)),$$

where

- $O((k-1)|P|^{k-1})$ is the worst-case running time complexity of performing $k-1$ consecutive grammar transformations,
- $C_P(|E|, |P|, k)$ is the worst-case running time complexity of generating a set of sequences achieving production coverage for G_k, and
- $C_T(|E|, |P|, k)$ is the worst-case running time complexity of inverse transforming these sequences to obtain test cases.

Generally, $C_P(|E|, |P|, k)$ is the dominant term. Although there is no detailed time complexity analysis for fast grammar-based test generation algorithms [146][119][176], the performance is generally polynomial in $|P|^k$, that is, $O(|P|^{ck})$ for some $c \geq 1$, where the number of productions in the corresponding k-Reg is $O(|P|^k)$ (See Section 5.1). For example, even if each production is covered a minimum number of times, the complexity becomes $O(|K|^{3k}) = O(|P|^{3k})$ [38].

Example 5.4 (Test Sets Generated Using Algorithm 5.4). When Algorithm 5.4 is executed on the 1-Reg in Figure 2.2 for $k = 1$, no transformation of the grammar is necessary. One can obtain the following set of test cases

{c1 c1 x1 c1 p1 c1 p1 x1 x1 p2 c1 p1 p1, x1 p2 x1 p2, c1 p1}

which achieves 2-sequence coverage. Furthermore, if $k = 2$ is used, the given 1-Reg is transformed once to obtain the 2-Reg in Figure 2.4, this 2-Reg is used to generate a sequence set and the elements of this set are inverse transformed to obtain test cases achieving 3-sequence coverage. The following is an example of test cases achieving 3-sequence coverage:

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

Naturally, during test execution, the corresponding basis event is used for each event, because the basis events represent the events as they are visible to user (See Definition 5.1).

6 Mutation Operators for Morphologically Different Models

Mutation operators are generally used to generate system models which are called mutants. Mutants contribute to testing process by serving as fault models; they can also be used in test evaluation or test generation. This chapter defines mutation operators for k-Regs for event-based testing and analyzes their properties based on [26][29][30][31]. Furthermore, the defined mutation operators are discussed in comparison to the grammar-based mutation operators [134][18].

In general, a mutant can be one of the following.

- A model where some behavior is missing
- A model where some behavior is extra
- A model where some behavior is missing and some behavior is extra
- A model which is equivalent to the original model

The fault types for event-based testing can be classified as *missing event* and *extra event*. In a *missing event fault*, an event cannot occur after or before performing a (possibly empty) sequence of events whereas it should; that is, the event is missing in some context. In an *extra event fault*, an event can occur after or before performing a (possibly empty) sequence of events whereas it should not; that is, the event is extra in some context.

Considering the elements of a k-Reg, the following mutation operators can be defined to model missing event and extra event faults. (Note that the term "terminal" instead of the term "k-sequence" to avoid a possible confusion between "insert sequence" and "insert k-sequence," and "omit sequence" and "omit k-sequence.")

- *Marking*: *Mark start (Ms)*, *mark non-start (Mns)*, *mark finish (Mf)* and *mark non-finish (Mnf)*.
- *Insertion*: *Insert sequence (Is)* and *insert terminal (It)*.
- *Omission*: *Omit sequence (Os)* and *omit terminal (Ot)*.

One can use mark nonstart, mark nonfinish, omit sequence and omit terminal operators to generate mutants modeling missing event faults; and mark start, mark finish, insert sequence and insert terminal operators can be used to generate mutants modeling extra event faults. These operators, except for insert terminal and omit terminal, inject small numbers of missing event or extra event faults. Although insert terminal and omit terminal may induce relatively large number of faults, these faults are local to specific parts of the model and therefore mostly different from the faults in another mutant of the same type. Also, each missing event fault modeled by an omit terminal mutant can be separately modeled by an omit sequence mutant and each extra event fault modeled by an insert terminal mutant can be separately modeled by an insert terminal mutants for which the mutation parameter size is properly limited.

Given a k-Reg G where $k \geq 2$, a k-Reg mutant $G' = (E', B', K', C', S, P')$ generated using one of the mutation operators defined in this chapter may not be a k-Reg by Definition 4.3. It may contain a production of the form $c(q) \rightarrow r\ c(r) \in P'$ where $q = q_1 \ldots q_k$ and $r = r_1 \ldots r_k$ such that

$$q_2 \ldots q_k \neq r_1 \ldots r_{k-1},$$

that is, ending (k-1)-sequence of q is not equal to the beginning (k-1)-sequence of r. This happens when new productions representing sequences that do not exist in the original model are inserted to perform the mutation. Fortunately, this mutant can still be treated as a k-Reg by ignoring $r_1 \ldots r_{k-1}$ (which would be left out from sequences in the grammar during inverse sequence transformation (See Definition 5.2)), because the primary interest lies in the relation between $q_1 \ldots q_k$ and r_k; that is, r_k follows $q_1 \ldots q_k$. For this reason, the definition of a k-Reg is relaxed while referring to mutants. Thus, the discussed mutation operators preserve the form (k-Reg) and the type (or regularity) of the grammar.

In the next three sections (Section 6.1 - Section 6.3), the following are discussed for each event-based mutation operator.

- Definition
- Algorithmic complexity
- Number of possible mutants
- Set of all CESs of each mutant
- Conditions for generation of useful, deterministic and non-equivalent mutants

In Section 6.4, event-based mutation operators defined for k-Regs are compared to grammar-based mutation operators [134][18].

6.1 Marking Operators

Marking operators are used to change the type of certain elements in the model. More precisely, they are defined to mark k-sequences as start, nonstart, finish, and nonfinish. Mark start and mark finish operators preserve usefulness of the given grammar, whereas mark nonstart and mark nonfinish operators may fail to do so. For the latter two, it is possible to outline usefulness preserving measures. However, they are skipped, because they involve performing additional marking operations on k-sequences in the grammar. For example, while marking a k-sequence e as a nonstart, one may have to mark e or a k-sequence other than e as a start k-sequence.

6.1.1 Mark Start

Mark start operator turns a given k-sequence into a start k-sequence. In this way, mutant models which have extra start start k-sequences can be constructed. The operator is defined as follows.

Definition 6.1 (Mark Start). Given a k-Reg $G = (E, B, K, C, S, P)$ and a k-sequence $e \in K$ such that $S \rightarrow e\ c(e) \notin P$, *mark start (Ms)* operator is defined as

$$Ms(G, e) = G' = (E, B, K, C, S, P')$$

where $P' = P \cup \{S \rightarrow e\ c(e)\}$.

As Definition 6.1 suggests, the operator assumes that k-sequence e to be marked as start is not already a start k-sequence. Therefore, it can be performed in $O(1)$ time by adding a single production $S \rightarrow e\ c(e)$. Checking whether k-sequence e is a start k-sequence can be performed in $O(|P|)$ steps by checking whether production $S \rightarrow e\ c(e)$ is in P.

In addition, a mark start mutant is also a k-Reg, and one can perform

$$|K| - s$$

mark start operations, where s is the number of start k-sequences.

A mark start mutant introduces new sequences to the set of all CESs. This is shown by Lemma 6.1 as follows.

Lemma 6.1 (Set of All CESs of a Mark Start Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ms(G, e)$. The set of all CESs of G' is given by

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

Proof: The proof follows from Definition 6.1.■

Using Lemma 6.1, Lemma 6.2 given below discusses the equivalence of a mark start mutant.

Lemma 6.2 (Equivalence of a Mark Start Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ms(G, e)$. Let

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

G' is not equivalent to G if and only if $d(X) \setminus d(Y) \neq \varnothing$.

Proof: $T_{CES}(G') = T_{CES}(G) \cup X$ (by Lemma 6.1). Hence, using Definition 4.8, $d(T_{CES}(G')) = d(T_{CES}(G))$ if and only if $d(X) \subseteq d(Y) \subseteq d(T_{CES}(G))$, which completes the proof.■

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

Theorem 6.1 (Usefulness of a Mark Start Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ms(G, e)$. G' is useful, if G is useful.

Proof: The proof follows from Definition 4.9 and Definition 6.1.■

Theorem 6.2 (Determinism of a Mark Start Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ms(G, e)$. G' is deterministic, if G is deterministic and there exists no $S \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$.

Proof: The proof follows from Definition 4.10 and Definition 6.1.■

Theorem 6.3 (Nonequivalence of a Mark Start Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ms(G, e)$. G' is not equivalent to G, if G is useful and there exists no $S \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$.

Proof: Let X and Y be the sets defined in Lemma 6.2.

- $T_{CES}(G) \neq \varnothing$ and $X \neq \varnothing$, since G is useful.
- $Y = \varnothing$, since there exists no $S \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$.

Thus, $d(X) \cap d(Y) = \varnothing$ and G' is not equivalent to G (Lemma 6.2).■

6.1.2 Mark Finish

Mark finish operator turns a given k-sequence into a finish k-sequence. Using this operator, mutant models which have extra finish k-sequences can be constructed. The operator is defined as follows.

Definition 6.2 (Mark Finish). Given a k-Reg $G = (E, B, K, C, S, P)$ and a k-sequence $e \in K$ such that $c(e) \rightarrow \varepsilon \notin P$, *mark finish (Mf)* operator is defined as

$$Mf(G, e) = G' = (E, B, K, C, S, P')$$

where $P' = P \cup \{c(e) \rightarrow \varepsilon\}$.

By Definition 6.2, the operator can be performed in $O(1)$ time assuming that k-sequence e is not already a finish k-sequence. One can check whether $c(e) \rightarrow \varepsilon$ is already in P in $O(|P|)$ time.

Note that a mark finish mutant is also a k-Reg. In addition, one can perform

$$|K| - f$$

mark finish operations, where f is the number of finish k-sequences.

The set of all CESs of the original model is a subset of the set of all CESs of a mark finish mutant, which is given by Lemma 6.3.

Lemma 6.3 (Set of All CESs of a Mark Finish Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mf(G, e)$. The set of all CESs of G' is given by

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

Proof: The proof follows from Definition 6.2.■

With the help of Lemma 6.3, the equivalence of a mark finish mutant can be discussed as follows.

Lemma 6.4 (Equivalence of a Mark Finish Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mf(G, e)$. Let

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

G' is not equivalent to G if and only if $d(X) \setminus d(Y) \neq \varnothing$.

Proof: $T_{CES}(G') = T_{CES}(G) \cup X$ (by Lemma 6.3). Thus, using Definition 4.8, $d(T_{CES}(G')) = d(T_{CES}(G))$ if and only if $d(X) \subseteq d(Y) \subseteq d(T_{CES}(G))$. This completes the proof.■

The following give sufficient conditions for usefulness, determinism and nonequivalence of a mark finish mutant.

Theorem 6.4 (Usefulness of a Mark Finish Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mf(G, e)$. G' is useful, if G is useful.

Proof: The proof follows from Definition 4.9 and Definition 6.2.■

Theorem 6.5 (Determinism of a Mark Finish Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mf(G, e)$. G' is deterministic, if G is deterministic.

Proof: The proof follows from Definition 4.10 and Definition 6.2.■

Theorem 6.6 (Nonequivalence of a Mark Finish Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mf(G, e)$. G' is not equivalent to G, if G is useful and deterministic.

Proof: Let X and Y be the sets defined in Lemma 6.4.

- $T_{CES}(G) \neq \varnothing$ and $X \neq \varnothing$, since G is useful.
- $d(X) \cap d(Y) = \varnothing$, since each m-derived start sequence in deterministic G has a unique basis (Theorem 4.4).

Hence, G' is not equivalent to G (Lemma 6.4).■

6.1.3 Mark Nonstart

Mark nonstart operator turns a given k-sequence into nonstart k-sequence, in order to obtain mutant models which have missing start k-sequences. The definition of the operator is given as follows.

Definition 6.3 (Mark Nonstart). Given a k-Reg $G = (E, B, K, C, S, P)$ and a k-sequence $e \in K$ such that $S \rightarrow e\ c(e) \in P$, *mark nonstart (Mns)* operator is defined as

$$Mns(G, e) = G' = (E, B, K, C, S, P')$$

where $P' = P \setminus \{S \rightarrow e\ c(e)\}$.

As Definition 6.3 suggests, the operator assumes that k-sequence e is already a start k-sequence. Regardless, the operator can be performed in $O(|P|)$ time by searching for production $S \rightarrow e\ c(e)$ in P, and removing it if it exists.

Furthermore, a mark nonstart mutant is also a k-Reg, and one can perform

$$s = |\{e|\ S \rightarrow e\ c(e) \in P\}|$$

mark nonstart operations.

The set of all CESs of a mark nonstart is a subset of the set of all CESs of the original model. This is shown by Lemma 6.5 as follows.

Lemma 6.5 (Set of All CESs of a Mark Nonstart Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mns(G, e)$. The set of all CESs of G' is given by

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

Proof: The proof follows from Definition 6.3.■

Lemma 6.5 shows that the mutant grammar may not be useful anymore. For example, if there exists no $S \rightarrow a\ c(a) \in P$ for some $a \neq e$, none of the k-sequences in the mutant grammar are useful; that is, $T_{CES}(G')$ is empty if all CESs in $T_{CES}(G)$ begins with e.

Using Lemma 6.5, Lemma 6.6 states the equivalence of a mark nonstart mutant as follows.

Lemma 6.6 (Equivalence of a Mark Nonstart Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mns(G, e)$. Let

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

G' is not equivalent to G if and only if $d(X) \setminus d(Y) \neq \varnothing$.

Proof: $T_{CES}(G') = T_{CES}(G) \setminus X$ (by Lemma 6.5). Hence, using Definition 4.8, $d(T_{CES}(G')) = d(T_{CES}(G))$ if and only if $d(X) \subseteq d(Y) \subseteq d(T_{CES}(G) \setminus X)$, which proves the theorem.■

Sufficient conditions of usefulness, determinism and nonequivalence of a mark nonstart mutant are given in the following.

Theorem 6.7 (Usefulness of a Mark Nonstart Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mns(G, e)$. G' is useful, if G is useful and there exists a start k-sequence $a \neq e$ such that $S \Rightarrow_G a\, c(a) \Rightarrow_G^* a\, x\, e\, c(e)\ (x \in E^*)$.

Proof: The proof follows from Definition 4.9 and Definition 6.3.■

Theorem 6.8 (Determinism of a Mark Nonstart Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mns(G, e)$. G' is deterministic, if G is deterministic.

Proof: The proof follows from Definition 4.10 and Definition 6.3.■

Theorem 6.9 (Nonequivalence of Mark Nonstart Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mns(G, e)$. G' is not equivalent to G, if G is useful and deterministic.

Proof: Let X and Y be the sets defined in Lemma 6.6.

- $T_{CES}(G) \neq \varnothing$ and $X \neq \varnothing$, since G is useful.
- $Y = \varnothing$, since G is deterministic.

Hence, $d(X) \cap d(Y) = \varnothing$ and G' is not equivalent to G (Lemma 6.6).■

6.1.4 Mark Nonfinish

Mark nonfinish operator turns a given k-sequence into a nonfinish k-sequence. This enables to construct mutant models which has missing finish k-sequences. The operator definition is given below.

Definition 6.4 (Mark Nonfinish). Given a k-Reg $G = (E, B, K, C, S, P)$ and a k-sequence $e \in K$ such that $c(e) \rightarrow \varepsilon \in P$, *mark nonfinish (Mnf)* operator is defined as

$$Mnf(G, e) = G' = (E, B, K, C, S, P')$$

where $P' = P \setminus \{c(e) \rightarrow \varepsilon\}$.

According to Definition 6.4, the operator assumes that k-sequence e is a finish k-sequence. Regardless of this assumption, the operator can be carried out in $O(|P|)$ time by searching for production $c(e) \rightarrow \varepsilon$ in P, and removing it, if it exists.

In addition, a mark nonfinish mutant is a k-Reg, and one can perform

$$f = |\{e|\ c(e) \rightarrow \varepsilon \in P\}|$$

mark nonfinish operations.

A mark nonfinish mutant removes some sequences from the set of all CESs. This is discussed by Lemma 6.7.

Lemma 6.7 (Set of All CESs of a Mark Nonfinish Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mnf(G, e)$. The set of all CESs of G' is given by

$$T_{CES}(G') = T_{CES}(G) \setminus \{x\ e|\ S \Rightarrow_G^* x\ e\ (x \in E^*)\}.$$

Proof: The proof follows from Definition 6.4.■

Lemma 6.7 shows that the mutant grammar may not be useful anymore. For example, if there exists no $c(a) \rightarrow \varepsilon \in P$ for some $a \neq e$, none of the k-sequences in the mutant grammar are useful; that is, $T_{CES}(G')$ is empty if all CESs in $T_{CES}(G)$ ends with e.

Using Lemma 6.7, Lemma 6.8 discusses the equivalence of a mark nonfinish mutant.

Lemma 6.8 (Equivalence of a Mark Nonfinish Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Mnf(G, e)$. Let

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

G' is not equivalent to G if and only if $d(X) \setminus d(Y) \neq \varnothing$.

Proof: $T_{CES}(G') = T_{CES}(G) \setminus X$ (by Lemma 6.7). Thus, using Definition 4.8, $d(T_{CES}(G')) = d(T_{CES}(G))$ if and only if $d(X) \subseteq d(Y) \subseteq d(T_{CES}(G) \setminus X)$. This proves the theorem.■

The following outline sufficient conditions for usefulness, determinism and nonequivalence of a mark nonstart mutant.

Theorem 6.10 (Usefulness of a Mark Nonfinish Mutant). Let $G = (E, B, K, C, S, P)$ be a k-Reg and $G' = Mnf(G, e)$. G' is useful, if G is useful and there exists a finish k-sequence $a \neq e$ such that $c(e) \Rightarrow_G^* x\, a\ (x \in E^*)$.

Proof: The proof follows from Definition 4.9 and Definition 6.4.■

Theorem 6.11 (Determinism of a Mark Nonfinish Mutant). Let $G = (E, B, K, C, S, P)$ be a k-Reg and $G' = Mnf(G, e)$. G' is deterministic, if G is deterministic.

Proof: The proof follows from Definition 4.10 and Definition 6.4.■

Theorem 6.12 (Nonequivalence of a Mark Nonfinish Mutant). Let $G = (E, B, K, C, S, P)$ be a k-Reg and $G' = Mnf(G, e)$. G' is not equivalent to G, G is useful and deterministic.

Proof: Let X and Y be the sets defined in Lemma 6.8.

- $T_{CES}(G) \neq \varnothing$ and $X \neq \varnothing$, since G is useful.
- $d(X) \cap d(Y) = \varnothing$, since each m-derived start sequence in deterministic G has a unique basis (Theorem 4.4).

Hence, G' is not equivalent to G (Lemma 6.8).■

6.2 Insertion Operators

Insertion operators are used to generate models that have additional functionality when compared to the original model; that is, the original model is a correct (sub) model of the mutant. This section outlines operators to add new sequences (of k-sequences) and terminals (k-sequences) to a given k-Reg model.

6.2.1 Insert Sequence

Insert sequence operator introduces a new terminal (or k-sequence) sequence to a given grammar by establishing a connection between two existing k-sequences. In this way, mutant models which contain different terminal sequences can be constructed. The operator is defined as follows.

Definition 6.5 (Insert Sequence). Given a k-Reg $G = (E, B, K, C, S, P)$ and a sequence (a, b) such that $a,b \in K$ and $c(a) \rightarrow b\ c(b) \notin P$, *insert sequence (Is)* operator is defined as

$$Is(G, (a, b)) = G' = (E, B, K, C, S, P')$$

where $P' = P \cup \{c(a) \rightarrow b\ c(b)\}$.

No usefulness preserving measures are given in Definition 6.5, because the operator does not violate usefulness, if the given grammar is already useful. Thus, it suffices to update the given grammar with a new production in order to include the intended sequence. Algorithm 6.1 outlines steps to perform the operator given in Definition 6.5.

Algorithm 6.1. Insert Sequence

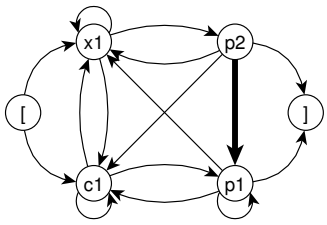
Input: $G = (E, B, K, C, S, P)$ – the input grammar
(a, b) where $a,b \in K$ and $c(a) \rightarrow b\ c(b) \notin P$ – the sequence to be inserted
Output: $G' = (E', B', K', C', S', P')$ – the updated grammar
$G' = G$
$P' = P' \cup \{c(a) \rightarrow b\ c(b)\}$ //Add the production for sequence (a, b)

Algorithm 6.1 has $O(1)$ worst case time complexity, since it is assumed that sequence to be inserted is not already in the given grammar. Checking whether a sequence is already in a given grammar can be performed in $O(|P|)$ time.

Example 6.1 (An Insert Sequence Mutant). Figure 6.1 shows the 1-Reg resulting from the insertion of sequence $(p2, p1)$ into the 1-Reg in Figure 2.2.

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

(a) 1-Reg.



(b) Directed graph visualization.

Figure 6.1. Insertion of sequence (p2, p1) to the 1-Reg in Figure 2.2.

Given a k-Reg, an insert sequence mutant is also a k-Reg, and one can perform

$$|K|^2 - |P| + (s+f)$$

insert sequence operations, where s is the number of start k-sequences and f is the number finish k-sequences. Note that $|K|^2$ is the number of all possible terminal sequences, and $(|P| - (s+f))$ is the number of existing terminal sequences.

The set of all CESs of an insert sequence mutant contains new sequences. Lemma 6.9 shows this considering different cases.

Lemma 6.9 (Set of All CESs of an Insert Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Is(G, (a, b))$. Let

- $M = \{x | S \Rightarrow_G^* x\ a\ c(a)\ (x \in E^*)\}$,
- $N = \{y | c(b) \Rightarrow_G^* y\ (y \in E^*)\}$ and
- $O = \{z | c(b) \Rightarrow_G^* z\ a\ c(a)\ (z \in E^*)\}$.

The set of all CESs of G' is given by

$$T_{CES}(G') = T_{CES}(G) \cup Mab(Oab)^*N, \text{ if } a \neq b, \text{ and}$$
$$T_{CES}(G') = T_{CES}(G) \cup Maa(a+Oa)^*N, \text{ if } a = b.$$

Proof: The proof follows from Definition 6.5 considering the cases $a \neq b$ and $a = b$.■

Making use of Lemma 6.9, the equivalence of an insert sequence mutant is discussed in Lemma 6.10.

Lemma 6.10 (Equivalence of an Insert Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Is(G, (a, b))$. Let

- $X = Mab(Oab)^*N$, if $a \neq b$; $X = Maa(a+Oa)^*N$, if $a = b$, and

- $Y = \{w \mid w \in T_{CES}(G)$ and w contains $a'b'$ such that $a', b' \in K$, $a' \neq a$ and $d(a') = d(a)$, and $b' \neq b$ and $d(b') = d(b)\} \subseteq T_{CES}(G)$,

where M, N and O are the sets defined in Lemma 6.9. G' is not equivalent to G if and only if $d(X) \setminus d(Y) \neq \varnothing$.

Proof: $T_{CES}(G') = T_{CES}(G) \cup X$ (by Lemma 6.9). Thus, using Definition 4.8, $d(T_{CES}(G')) = d(T_{CES}(G))$ if and only if $d(X) \subseteq d(Y) \subseteq d(T_{CES}(G))$. This completes the proof.■

In the following, sufficient conditions for usefulness, determinism and nonequivalence of an insert sequence mutant are given.

Theorem 6.13 (Usefulness of an Insert Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Is(G, (a, b))$. G' is useful, if G is useful.

Proof: The proof follows from Definition 4.9 and Definition 6.5.■

Theorem 6.14 (Determinism of an Insert Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Is(G, (a, b))$. G' is deterministic, if the following conditions hold:

1. G is deterministic.
2. There exists no $c(a) \rightarrow b'\, c(b') \in P$ such that $b' \neq b$ and $d(b') = d(b)$.

Proof: The proof follows from Definition 4.10 and Definition 6.5.■

Theorem 6.15 (Nonequivalence of an Insert Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Is(G, (a, b))$. G' is not equivalent to G, if the following conditions hold.

1. G is useful.
2. G is deterministic.
3. There exists no $c(a) \rightarrow b'\, c(b') \in P$ such that $b' \neq b$ and $d(b') = d(b)$.

Proof: Let X and Y be the sets defined in Lemma 6.10.

- $T_{CES}(G) \neq \varnothing$ and $X \neq \varnothing$, since G is useful.
- $d(X) \cap d(Y) = \varnothing$, since each m-derived start sequence in deterministic G has a unique basis (Theorem 4.4).

Hence, G' is not equivalent to G (Lemma 6.10).■

6.2.2 Insert Terminal

Insert terminal operator adds a new k-sequence to a given grammar, possibly establishing its connections to the existing k-sequences. In this way, mutant models which contain k-sequences with different contexts can be created. The operator is defined as follows.

Definition 6.6 (Insert Terminal). Given a k-Reg $G = (E, B, K, C, S, P)$, a k-sequence e such that $e \notin K$ and $d(e) \in B^k$, a set of ingoing sequences $U = \{(a, e) | a \in \{a_1, \ldots, a_m\} \subseteq K\}$ and a set of outgoing sequences $V = \{(e, b) | b \in \{b_1, \ldots, b_n\} \subseteq K \cup \{e\}\}$, *insert terminal (It)* operator is defined as

$$It(G, e, U, V) = G' = (E, B, K', C', S, P'),$$

where $K' = K \cup \{e\}$, $C' = C \cup \{c(e)\}$, and, for $Q = P \cup \{c(e) \rightarrow a\ c(a) | (a, e) \in U\} \cup \{c(b) \rightarrow e\ c(e) | (e, b) \in V\}$:

- If usefulness preservation is not required,
  - $P' = Q$.
- If usefulness preservation is required, let $U' = \{c(e) \rightarrow \varepsilon | V = \varnothing$ or $V = \{(e, e)\}\}$ and $V' = \{S \rightarrow e\ c(e) | U = \varnothing\}$,
  - $P' = Q \cup (U' \cup V')$.

As Definition 6.6 suggests, insertion of a new k-sequence requires adding a new k-sequence, a new context, and new productions for the sequences. Nevertheless, if usefulness of the new k-sequence cannot be established, insertion of additional productions may be required. The steps to update the given grammar with the insertion of a k-sequence are given in Algorithm 6.2 including usefulness preserving measures.

Algorithm 6.2 assumes that $e \notin K$, which can be checked in $O(|K|)$ time. Therefore, running time complexity of Algorithm 6.2 is given by $O(m+n) = O(|K|)$, where m is the number of sequences of the form "$(\ldots, e)$", that is, ingoing sequences, and n is the number of sequences of the form "$(e, \ldots)$", that is, outgoing sequences, to be inserted. Note that (e, e), that is, the looping sequence, is considered to be an outgoing sequence and, therefore, $(m+n) \leq 2|K|+1 = 2(|C|-1)+1$.

Briefly, Algorithm 6.2 adds a new k-sequence and its corresponding context. Later, for each sequence to be inserted, a new production is added. To be sure that the resulting grammar is useful, one needs to check two cases.

Algorithm 6.2. Insert Terminal

Input: $G = (E, B, K, C, S, P)$ – the input grammar
$e \notin K$ where $d(e) \in B^k$ – the k-sequence to be inserted
$U = \{(a, e) | a \in \{a_1, \ldots, a_m\} \subseteq K\}$ – the ingoing sequences to be inserted
$V = \{(e, b) | b \in \{b_1, \ldots, b_n\} \subseteq K \cup \{e\}\}$ – the outgoing sequences to be inserted
Output: $G' = (E', B', K', C', S', P')$ – the updated grammar

```
G' = G
K' = K' ∪ {e}      //Add the new k-sequence e
C' = C' ∪ {c(e)}   //Add new context c(e)
for each (a, e) ∈ U do
    P' = P' ∪ {c(a) → e(e)}   //Add the production for sequence (a, e)
endfor
for each (e, b) ∈ V do
    P' = P' ∪ {c(e) → b c(b)}   //Add the production for sequence (e, b)
endfor
if m < 1 then
    P' = P' ∪ {S → e c(e)}   //Usefulness of e (Part 1: Mark e as start)
endif
if n < 1 or (n = 1 and b1 = e) then
    P' = P' ∪ {c(e) → ε}   //Usefulness of e (Part 2: Mark e as finish)
endif
```

Let G be a useful k-Reg. After insertion of a new k-sequence e, the resulting grammar G' may not be useful due to the following.

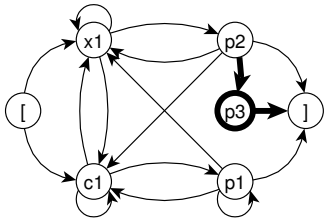
1. If the set of ingoing sequences is empty (that is, $U = \varnothing$), no derivations of the form $S \Rightarrow_{G'}^{*} x\ e\ c(e)$ $(x \in E^{*})$ exist; that is, derivations starting from S do not go through $c(e)$. Therefore, production $S \rightarrow e\ c(e)$ should be added for usefulness.
2. If the set of outgoing sequences is empty or contains only the looping sequence (that is, $V = \varnothing$ or $V = \{(e, e)\}$), a derivation of the form $c(e) \Rightarrow_{G'}^{*} x$ $(x \in E^{*})$ does not exist; that is, derivations starting from $c(e)$ do not terminate. Thus, production $c(e) \rightarrow \varepsilon$ should be added for usefulness.

In addition, if the given grammar G is not useful, (1) or (2) are not guaranteed to preserve the usefulness of the resulting grammar, but they still make inserted k-sequence e useful.

Example 6.2 (An Insert Terminal Mutant). Figure 6.2 shows the 1-Reg resulting from the insertion of 1-sequence *p3* together with $\{(p2, p3)\}$ into the 1-Reg in Figure 2.2 where $c(p3) \rightarrow \varepsilon$ is also inserted to preserve usefulness.

S → c1 c(c1) | x1 c(x1)
c(c1) → c1 c(c1) | x1 c(x1) | p1 c(p1)
c(x1) → c1 c(c1) | x1 c(x1) | p2 c(p2)
c(p1) → c1 c(c1) | x1 c(x1) | p1 c(p1) | ε
c(p2) → c1 c(c1) | x1 c(x1) | ε **| p3 c(p3)**
c(p3) → ε

(a) 1-Reg.



(b) Directed graph visualization.

Figure 6.2. Insertion of p3 together with {(p2, p3)} to the 1-Reg in Figure 2.2.

Given a k-Reg, an insert terminal mutant is also a k-Reg. Furthermore, insertion of a single terminal can be performed in

$$2^{2|K|+1}$$

ways by selecting the sets of ingoing and outgoing sequences differently. Thus, without some limitation, the operator may produce quite a large number of mutants in general. For example, by creating a mutant using a new contexted version of each basis k-sequence in B^k with one ingoing sequence and one outgoing sequence, one can generate

$$|K|\ |B|^k\ (|K|+1)$$

mutants. Using such limitations allows one to obtain insert terminal mutants which have fewer and separate changes, and tend to be more beneficial for test generation.

The set of all CESs of an insert terminal mutant may change depending on whether usefulness preserving measures are performed or not, which is given in Lemma 6.11.

Lemma 6.11 (Set of All CESs of an Insert Terminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = It(G, e, U, V)$ where $U = \{(a, e) | a \in \{a_1, \ldots, a_m\} \subseteq K\}$ and $V = \{(e, b) | b \in \{b_1, \ldots, b_n\} \subseteq K \cup \{e\}\}$. Let

- $M = \{x | S \Rightarrow_G^* x\ c(a)\ (x \in E^*)$ for $a = a_1, \ldots, a_m\}$, if $U \neq \varnothing$,
  $M = \{\varepsilon\}$, if $U = \varnothing$ and $S \rightarrow e\ c(e)$ is inserted to for usefulness of e, and
  $M = \varnothing$, if $U = \varnothing$ and $S \rightarrow e\ c(e)$ is not inserted to for usefulness of e.
- $N = \{y | b\ c(b) \Rightarrow_G^* y\ (y \in E^*)$ for $b = b_1, \ldots, b_n\ (b \neq e)\}$, if $V \neq \varnothing$ and $V \neq \{(e, e)\}$,
  $N = \{\varepsilon\}$, if $V = \varnothing$ or $V = \{(e, e)\}$, and $c(e) \rightarrow \varepsilon$ is inserted for usefulness of e, and

$N = \varnothing$, if $V = \varnothing$ or $V = \{(e, e)\}$, and $c(e) \rightarrow \varepsilon$ is not inserted for usefulness of e.

- $O = \{z |\ b\ c(b) \Rightarrow_G^* z\ c(a)\ (z \in E^*)$ for $a = a_1, \ldots, a_m$ and $b = b_1, \ldots, b_n\}$.

The set of all CESs of G' is given by

$$T_{CES}(G') = T_{CES}(G) \cup Me(Oe)^*N, \text{ if } (e, e) \notin V, \text{ and}$$

$$T_{CES}(G') = T_{CES}(G) \cup Me(e+Oe)^*N, \text{ if } (e, e) \in V.$$

Proof: The proof follows from Definition 6.6 considering the conditions for performing measures for usefulness of e, if they are required.■

Making use of Lemma 6.11, equivalence of an insert terminal mutant can be discussed as in Lemma 6.12.

Lemma 6.12 (Equivalence of an Insert Terminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = It(G, e, U, V)$ where $U = \{(a, e) |\ a \in \{a_1, \ldots, a_m\} \subseteq K\}$ and $V = \{(e, b) |\ b \in \{b_1, \ldots, b_n\} \subseteq K \cup \{e\}\}$. Let

- $X = Me(Oe)^*N$, if $(e, e) \notin V$, and $X = Me(e+Oe)^*N$, if $(e, e) \in V$, and
- $Y = \{w |\ w \in T_{CES}(G)$ and w contains e' where $e' \in K$, $e' \neq e$ and $d(e') = d(e)\} \subseteq T_{CES}(G)$,

where M, N and O are the sets defined in Lemma 6.11. G' is not equivalent to G if and only if $d(X) \setminus d(Y) \neq \varnothing$.

Proof: $T_{CES}(G') = T_{CES}(G) \cup X$ (by Lemma 6.11). Thus, using Definition 4.8, $d(T_{CES}(G')) = d(T_{CES}(G))$ if and only if $d(X) \subseteq d(Y) \subseteq d(T_{CES}(G))$. This completes the proof.■

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

Theorem 6.16 (Usefulness of an Insert Terminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = It(G, e, U, V)$ where $U = \{(a, e) |\ a \in \{a_1, \ldots, a_m\} \subseteq K\}$ and $V = \{(e, b) |\ b \in \{b_1, \ldots, b_n\} \subseteq K \cup \{e\}\}$. G' is useful, if the following conditions hold.

1. G is useful.
2. $S \rightarrow e\ c(e)$ is inserted for usefulness of e, if $U = \varnothing$.
3. $c(e) \rightarrow \varepsilon$ is inserted for usefulness of e, if $V = \varnothing$ or $V = \{(e, e)\}$.

Proof: The proof follows from Definition 4.9 and Definition 6.6.■

Theorem 6.17 (Determinism of an Insert Terminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = It(G, e, U, V)$ where $U = \{(a, e) | a \in \{a_1, \ldots, a_m\} \subseteq K\}$ and $V = \{(e, b) | b \in \{b_1, \ldots, b_n\} \subseteq K \cup \{e\}\}$. G' is deterministic, if the following conditions hold.

1. G is deterministic.
2. There exists no $c(a) \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$ for $a = a_1, \ldots, a_m$.
3. There exist no b_i, b_j such that $b_i \neq b_j$ and $d(b_i) = d(b_j)$, for some $i, j \in \{1, \ldots, n\}$.
4. $S \rightarrow e\ c(e)$ is not inserted for usefulness of e, if there exists $S \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$.

Proof: The proof follows from Definition 4.10 and Definition 6.6.■

Theorem 6.18 (Nonequivalence of an Insert Terminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = It(G, e, U, V)$ where $U = \{(a, e) | a \in \{a_1, \ldots, a_m\} \subseteq K\}$ and $V = \{(e, b) | b \in \{b_1, \ldots, b_n\} \subseteq K \cup \{e\}\}$. G' is not equivalent to G, if the following conditions hold.

1. G is useful.
2. G is deterministic.
3. $c(e) \rightarrow \varepsilon$ is inserted for usefulness of e, if $V = \varnothing$ or $V = \{(e, e)\}$.
4. $S \rightarrow e\ c(e)$ is inserted for usefulness of e, if $U = \varnothing$.
5. $U \neq \varnothing$, if there exists $S \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$.
6. There exists no $c(a) \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$ for $a = a_1, \ldots, a_m$.

Proof: Let X and Y be the sets defined in Lemma 6.12.

- $T_{CES}(G) \neq \varnothing$, since G is useful.
- $X \neq \varnothing$, since G is useful, $U \neq \varnothing$, if there exists $S \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$, $S \rightarrow e\ c(e)$ is inserted for usefulness of e, if $U = \varnothing$, and $c(e) \rightarrow \varepsilon$ is inserted for usefulness of e, if $V = \varnothing$ or $V = \{(e, e)\}$.
- $d(X) \cap d(Y) = \varnothing$, since each m-derived start sequence in deterministic G has a unique basis (Theorem 4.4) and there exists no $c(a) \rightarrow e'\ c(e') \in P$ such that $e' \neq e$ and $d(e') = d(e)$ for $a = a_1, \ldots, a_m$.

Hence, G' is not equivalent to G (Lemma 6.12).■

6.3 Omission Operators

Application of an omission operator almost always yields a model that is sub model of the original model. This section outlines operators to remove existing sequences (of k-sequences) and terminals (k-sequences) from a given k-Reg model.

6.3.1 Omit Sequence

Omit sequence operator removes an existing connection between two terminals (or k-sequences) in a given grammar. Doing so, mutants with missing sequence can be created. The operator is defined as follows.

Definition 6.7 (Omit Sequence). Given a k-Reg $G = (E, B, K, C, S, P)$ and a sequence (a, b) such that $a, b \in K$ and $c(a) \rightarrow b\ c(b) \in P$, *omit sequence (Os)* operator is defined as

$$Os(G, (a, b)) = G' = (E, B, K, C, S, P')$$

where, for $Q = P \setminus \{c(a) \rightarrow b\ c(b)\}$:

- If usefulness preservation is not required,
  - $P' = Q$.
- If usefulness preservation is required, let $H = (E, B, K, C, S, Q)$, $U' = \{c(a) \rightarrow \varepsilon \mid a \text{ is not useful in } H\}$ and $V' = \{S \rightarrow b\ c(b) \mid b \text{ is not useful in } H\}$,
  - $P' = Q \cup (U' \cup V')$.

Definition 6.7 includes optional usefulness preserving measures, because, after removing the production corresponding to the sequence to be omitted, the resulting grammar may not be useful. Note that these measures aim to preserve usefulness of a and b, if they are already useful in the given grammar. Therefore, they preserve usefulness of the resulting grammar, if the given grammar is already useful. Algorithm 6.3 outlines the steps for omit sequence operator where measures to preserve usefulness are performed by default.

Algorithm 6.3 runs in $O(|K|+|P|)$ time because removal and testing membership of a production can both be performed in $O(|P|)$ steps and checking the existence of derivations of the form $c(a) \Rightarrow_{G'}^{*} y$ $(y \in E^{*})$ and $S \Rightarrow_{G'}^{*} x\ b\ c(b)$ $(x \in E^{*})$ can be performed in $O(|K|+|P|)$ time.

Algorithm 6.3. Omit Sequence

Input: $G = (E, B, K, C, S, P)$ – the input grammar
(a, b) where $a,b \in K$ and $c(a) \rightarrow b\ c(b) \in P$ – the sequence to be omitted
Output: $G' = (E', B, K', C', S', P')$ – the updated grammar
$G' = G$
$P' = P \setminus \{c(a) \rightarrow b\ c(b)\}$ //Remove the production for sequence (a, b)
if there exists no $c(a) \Rightarrow_{G'}^{*} y\ (y \in E^{*})$ **then**
$P' = P' \cup \{c(a) \rightarrow \varepsilon\}$ //Usefulness of a (Part 1: Mark a as finish)
endif
if there exists no $S \Rightarrow_{G'}^{*} x\ b\ c(b)\ (x \in E^{*})$ **then**
$P' = P' \cup \{S \rightarrow b\ c(b)\}$ //Usefulness of b (Part 2: Mark b as start)
endif

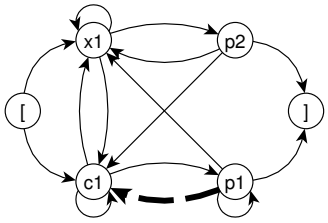
Let G be a useful k-Reg. If $a \neq b$, after the removal of production $c(a) \rightarrow b\ c(b)$, the resulting grammar G' may not be useful due to the following.

1. a is not useful in G'. In this case, although there exists $S \Rightarrow_{G'}^{*} x\ a\ c(a)\ (x \in E^{*})$, $c(a) \Rightarrow_{G'}^{*} y\ (y \in E^{*})$ does not exist; that is, derivations starting from S and going through $c(a)$ do not terminate. To preserve the usefulness, a new production $c(a) \rightarrow \varepsilon$ can be added to the grammar.
2. b is not useful in G'. More precisely, although there exists $c(b) \Rightarrow_{G'}^{*} y\ (y \in E^{*})$, $S \Rightarrow_{G'}^{*} xbc(b)\ (x \in E^{*})$ does not exist; that is, derivations starting from S do not go through $c(b)$, though derivations starting from $c(b)$ terminate. In this case, to preserve the usefulness, a new production $S \rightarrow b\ c(b)$ can be added.

Consequently, if the given grammar G is not useful, or a or b are not useful in G, (1) or (2) are not guaranteed to preserve the usefulness of the resulting grammar, or the usefulness of a or b.

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

(a) 1-Reg.



(b) Directed graph visualization.

Figure 6.3. Omission of sequence (p1, c1) from the 1-Reg in Figure 2.2.

Example 6.3 (An Omit Sequence Mutant). Figure 6.3 shows the 1-Reg resulting from the omission of sequence *(p1, c1)* from the 1-Reg in Figure 2.2.

Given a k-Reg, an omit sequence mutant is also a k-Reg. Furthermore, one can perform

$$|P| - (s+f)$$

omit sequence operations, where s is the number of start k-sequences and f is the number of finish k-sequences.

The set of all CESs of an omit sequence mutant may change depending on whether usefulness preserving measures are performed or not, which is given in Lemma 6.13.

Lemma 6.13 (Set of All CESs of an Omit Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Os(G, (a, b))$. Let

- $X = \{w | S \Rightarrow_G^* x\ a\ b\ c(b) \Rightarrow_G^* w\ (x,w \in E^*)\}$,
- $M = \{x\ a | S \Rightarrow_G^* x\ a\ c(a)\ (x \in E^*)$ where x does not contain $ab\}$, if $c(a) \rightarrow \varepsilon$ is inserted for usefulness of a, and $M = \varnothing$, otherwise, and
- $N = \{b\ y | c(b) \Rightarrow_G^* y\ (y \in E^*)$ where y does not contain $ab\}$, if $S \rightarrow b\ c(b)$ is inserted for usefulness of b, and $N = \varnothing$, otherwise.

The set of all CESs of G' is given by

$$T_{CES}(G') = (T_{CES}(G) \setminus X) \cup (M \cup N).$$

Proof: The proof follows from Definition 6.7 considering the measures for usefulness of a and b.■

Using Lemma 6.13, Lemma 6.14 discusses the equivalence of an omit sequence mutant as follows.

Lemma 6.14 (Equivalence of an Omit Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Os(G, (a, b))$. Let

- $Y = \{w | w \in T_{CES}(G)$ such that w does not contain ab but it contains $a'b'$ where $a' \neq a$ and $d(a') = d(a)$, and $b' \neq b$ and $d(b') = d(b)\} \subseteq (T_{CES}(G) \setminus X)$ and
- $Z = \{w | w \in T_{CES}(G)$ such that w does not contain ab, and either w ends with $a' \neq a$ where $d(a') = d(a)$ or w begins with $b' \neq b$ where $d(b') = d(b)$ or both$\} \subseteq (T_{CES}(G) \setminus X)$,

where X, M and N are the sets defined in Lemma 6.13. G' is not equivalent to G if and only if $d(M \cup N) \setminus d(Z) \neq \varnothing$ or $(d(X) \setminus d(Y)) \setminus d(M \cup N) \neq \varnothing$.

Proof: $T_{CES}(G') = (T_{CES}(G) \setminus X) \cup (M \cup N)$ (by Lemma 6.13). Thus, by Definition 4.8, $d(T_{CES}(G')) = d(T_{CES}(G))$ if and only if $d(M \cup N) \subseteq d(Z) \subseteq d(T_{CES}(G))$ and $d(X) \setminus d(Y) \subseteq d(M \cup N)$. This completes the proof.■

Sufficient conditions for usefulness, determinism and nonequivalence of an omit sequence mutant are given in the following.

Theorem 6.19 (Usefulness of an Omit Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Os(G, (a, b))$. G' is useful, if the following conditions hold.

1. G is useful.
2. $c(a) \rightarrow \varepsilon$ is inserted for usefulness of a, if there exists no derivation of the form $c(a) \Rightarrow_G^* x\ (x \in E^*)$ where x does not begin with b.
3. $S \rightarrow b\ c(b)$ is inserted for usefulness of b, if there exists no derivation of the form $S \Rightarrow_G^* x\ b\ c(b)\ (x \in E^*)$ where x does not end with a.

Proof: The proof follows from Definition 4.9 and Definition 6.7.■

Theorem 6.20 (Determinism of an Omit Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Os(G, (a, b))$. G' is deterministic, if the following conditions hold.

1. G is deterministic.
2. $S \rightarrow b\ c(b)$ is not inserted for usefulness of b, if there exists $S \rightarrow b'\ c(b') \in P$ such that $b' \neq b$ and $d(b') = d(b)$.

Proof: The proof follows from Definition 4.10 and Definition 6.7.■

Theorem 6.21 (Nonequivalence of an Omit Sequence Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Os(G, (a, b))$. G' is not equivalent to G, if the following conditions hold.

1. G is useful.
2. G is deterministic.
3. $S \rightarrow b\ c(b)$ is not inserted for usefulness of b, if there exists $S \rightarrow b'\ c(b') \in P$ such that $b' \neq b$ and $d(b') = d(b)$.

Proof: Let X, M and N be the sets defined in Lemma 6.13, and Y and Z be the sets defined in Lemma 6.14.

- $T_{CES}(G) \neq \varnothing$ and $X \neq \varnothing$, since G is useful.
- $d(X) \cap d(Y) = \varnothing$, since each CES in deterministic G has a unique basis (Corollary 4.2).

- $d(M \cup N) \cap d(Z) = \varnothing$, because
  - $d(M) \cap d(Z) = \varnothing$, since each m-derived start sequence in deterministic G has a unique basis (Theorem 4.4), and
  - $d(N) \cap d(Z) = \varnothing$, since $S \rightarrow b\ c(b)$ is not inserted for usefulness of b, if there exists $S \rightarrow b'\ c(b') \in P$ such that $b' \neq b$ and $d(b') = d(b)$.

Since $d(X) \cap d(Y) = \varnothing$, $d(X) \cap d(Z) = \varnothing$. This leads to $d(X) \cap d(M \cup N) = \varnothing$, because $d(M \cup N) \cap d(Z) = \varnothing$. Hence, G' is not equivalent to G (Lemma 6.14).■

6.3.2 Omit Terminal

Omit terminal operator removes an existing k-sequence from the given grammar by severing its connections to the other k-sequences. Thus, the mutant grammar does not contain this k-sequence. The operator defined as follows.

Definition 6.8 (Omit Terminal). Given a k-Reg $G = (E, B, K, C, S, P)$ and a k-sequence $e \in K$, *omit terminal (Ot)* operator is defined as

$$Ot(G, e) = G' = (E, B, K', C', S, P')$$

where $K' = K \setminus \{e\}$, $C' = C \setminus \{c(e)\}$, and, for $Q = P \setminus (\{c(e) \rightarrow b\ c(b) \in P \mid b \in K\} \cup \{c(a) \rightarrow e\ c(e) \in P \mid a \in K \setminus \{e\}\} \cup \{S \rightarrow e\ c(e), c(e) \rightarrow \varepsilon\})$:

- If usefulness preservation is not required,
  - $P' = Q$.
- If usefulness preservation is required, let $H = (E', B, K', C', S, Q)$, $U' = \{c(a) \rightarrow \varepsilon \mid c(a) \rightarrow e\ c(e) \in P$ and a is not useful in $H\}$ and $V' = \{S \rightarrow b\ c(b) \mid c(e) \rightarrow b\ c(b) \in P$ and b is not useful in $H\}$,
  - $P' = Q \cup (U' \cup V')$.

Omission of an existing k-sequence requires removing all the productions related to this k-sequence, the context related to this k-sequence and the k-sequence itself. In Definition 6.8, additional measures are carried out to preserve usefulness of other k-sequences, which may be affected by the removed productions. As usual, these measures aim to preserve usefulness of such k-sequences, if they are already useful in the given grammar. Therefore, the usefulness of the resulting grammar is preserved, if the given grammar is already useful. Algorithm 6.4 outlines the steps for omit terminal operator where measures to preserve usefulness are performed by default.

Algorithm 6.4. Omit Terminal

Input: $G = (E, B, K, C, S, P)$ – the input grammar
$e \in K$ – the k-sequence to be omitted
Output: $G' = (E', B', K', C', S', P')$ – the updated grammar
$G' = G$
$P' = P' \setminus \{S \to e\ c(e),\ c(e) \to \varepsilon\}$ //Remove productions (Mark e as nonstart and nonfinish)
$U' = \varnothing$, $V' = \varnothing$
for each $c(a) \to e\ c(e) \in P'$ and $a \in K' \setminus \{e\}$ **do**
  $P' = P' \setminus \{c(a) \to e\ c(e)\}$ //Remove the production for sequence (a, e)
endfor
for each $c(e) \to b\ c(b) \in P'$ and $b \in K'$ **do**
  $P' = P' \setminus \{c(e) \to b\ c(b)\}$ //Remove the production for sequence (e, b)
endfor
for each $c(a) \to e\ c(e) \in P$ and $a \in K \setminus \{e\}$ **do**
  if there exists no $c(a) \Rightarrow_{G'}^{*} y\ (y \in E^{*})$ **then**
    $U' = U' \cup \{c(a) \to \varepsilon\}$ //Usefulness of a (Mark a as finish)
  endif
endfor
for each $c(e) \to b\ c(b) \in P$ and $b \in K$ **do**
  if $b \neq e$ and there exists no $S \Rightarrow_{G'}^{*} x\ b\ c(b)\ (x \in E^{*})$ **then**
    $V' = V' \cup \{S \to b\ c(b)\}$ //Usefulness of b (Mark b as start)
  endif
endfor
$P' = P' \cup (U' \cup V')$ //Productions for usefulness
$C' = C' \setminus \{c(e)\}$ //Remove context $c(e)$
$K' = K' \setminus \{e\}$ //Remove k-sequence e

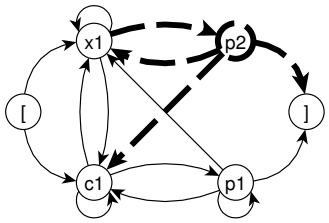
Algorithm 6.4 assumes $e \in K$, which can be checked in $O(|K|)$ time. Therefore, the algorithm terminates in $O((m+n)\ (|K|+|P|)) = O(|K|\ (|K|+|P|))$ number of steps, where m is the number of sequences of the form "$(\ldots, e)$", that is, ingoing sequences, and n is the number of sequences of the form "$(e, \ldots)$", that is, outgoing sequences. Note that (e, e), that is the looping sequence, is considered to be an outgoing sequence and, thus, $(m+n) \leq 2|K|-1 = 2(|C|-1)-1$.

Algorithm 6.4 first removes productions $S \to e\ c(e)$ and $c(e) \to \varepsilon$ from P, if they exist. Later, it removes productions related to the sequences ingoing to and outgoing from e, and then performs operations to preserve usefulness. Finally, context $c(e)$, and k-sequence e are removed from C and K, respectively.

Example 6.4 (An Omit Terminal Mutant). Figure 6.4 shows the 1-Reg resulting from the omission of 1-sequence *p2* from the 1-Reg in Figure 2.2.

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

(a) 1-Reg.



(b) Directed graph visualization.

Figure 6.4. Omission of p2 from the 1-Reg in Figure 2.2.

Given a k-Reg, an omit terminal mutant is also a k-Reg. Furthermore, one can perform

$$|K| = |C| - 1$$

omit terminal operations.

The set of all CESs of an omit sequence mutant may change depending on whether usefulness preserving measures are performed or not, which is given in Lemma 6.15.

Lemma 6.15 (Set of All CESs of an Omit Treminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ot(G, e)$. Let

- $X = \{w | S \Rightarrow_G^* x\, e\, c(e) \Rightarrow_G^* w\ (x,w \in E^*)\}$,
- $M = \{x\, a | S \Rightarrow_G^* x\, a\, c(a)\ (x \in E^*)$ where x does not contain e, $c(a) \rightarrow e\ c(e) \in P$ and $c(a) \rightarrow \varepsilon$ is inserted for usefulness of $a\}$, and
- $N = \{b\, y | c(b) \Rightarrow_G^* y\ (y \in E^*)$ where y does not contain e, $c(e) \rightarrow b\ c(b) \in P$ and $S \rightarrow b\ c(b)$ is inserted for usefulness of $b\}$.

The set of all CESs of G' is given by

$$T_{CES}(G') = (T_{CES}(G) \setminus X) \cup (M \cup N).$$

Proof: The proof follows from Definition 6.8 considering measures for usefulness, if they are required.■

Using Lemma 6.15, Lemma 6.16 discusses the equivalence of an omit terminal mutant as follows.

Lemma 6.16 (Equivalence of an Omit Terminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ot(G, e)$. Let

- $Y = \{w | w \in T_{CES}(G)$ such that w does not contain e but it contains e' where $e' \neq e$ and $d(e') = d(e)\} \subseteq (T_{CES}(G) \setminus X)$ and

- $Z = \{w |\ w \in T_{CES}(G)$ such that w does not contain e, and either w ends with $a' \neq a$ where $d(a') = d(a)$ and $c(a) \rightarrow \varepsilon$ is inserted for usefulness of a or w begins with $b' \neq b$ where $d(b') = d(b)$ and $S \rightarrow b\ c(b)$ is inserted for usefulness of b or both$\} \subseteq (T_{CES}(G) \setminus X)$,

where X, M and N are the sets defined in Lemma 6.15. G' is not equivalent to G if and only if $d(M \cup N) \setminus d(Z) \neq \varnothing$ or $(d(X) \setminus d(Y)) \setminus d(M \cup N) \neq \varnothing$.

Proof: $T_{CES}(G') = (T_{CES}(G) \setminus X) \cup (M \cup N)$ (by Lemma 6.15). Thus, using Definition 4.8, $d(T_{CES}(G')) = d(T_{CES}(G))$ if and only if $d(M \cup N) \subseteq d(Z) \subseteq d(T_{CES}(G))$ and $d(X) \setminus d(Y) \subseteq d(M \cup N)$. This completes the proof.■

Sufficient conditions for usefulness, determinism and nonequivalence of an omit terminal mutant are given in the following.

Theorem 6.22 (Usefulness of an Omit Terminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ot(G, e) = (E', B, K', C', S, P')$. Let U' and V' be the sets defined in Definition 6.8. G' is useful, if the following conditions hold.

1. G is useful.
2. U' is included in P'; that is, $U' \subseteq P'$.
3. V' is included in P'; that is, $V' \subseteq P'$.

Proof: The proof follows from Definition 4.9 and Definition 6.8.■

Theorem 6.23 (Determinism of an Omit Terminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ot(G, e)$. G' is deterministic, if the following conditions hold.

1. G is deterministic.
2. For each $b \in K$ such that $c(e) \rightarrow b\ c(b) \in P$, $S \rightarrow b\ c(b)$ is not inserted for usefulness of b, if there exists $S \rightarrow b'\ c(b') \in P$ such that $b' \neq b$ and $d(b') = d(b)$.

Proof: The proof follows from Definition 4.10 and Definition 6.8.■

Theorem 6.24 (Nonequivalence of an Omit Terminal Mutant). Given a k-Reg $G = (E, B, K, C, S, P)$ and $G' = Ot(G, e)$. G' is not equivalent to G, if the following conditions hold.

1. G is useful.
2. G is deterministic.

3. For each $b \in K$ such that $c(e) \rightarrow b\ c(b) \in P$, $S \rightarrow b\ c(b)$ is not inserted for usefulness of b, if there exists $S \rightarrow b'\ c(b') \in P$ such that $b' \neq b$ and $d(b') = d(b)$.

Proof: Let X, M and N be the sets defined in Lemma 6.15, and Y and Z be the sets defined in Lemma 6.16.

- $T_{CES}(G) \neq \varnothing$ and $X \neq \varnothing$, since G is useful.
- $d(X) \cap d(Y) = \varnothing$, since each CES in deterministic G has a unique basis (Corollary 4.2).
- $d(M \cup N) \cap d(Z) = \varnothing$, because
    - $d(M) \cap d(Z) = \varnothing$, since each m-derived start sequence in deterministic G has a unique basis (Theorem 4.4), and
    - $d(N) \cap d(Z) = \varnothing$, for each $b \in K$ such that $c(e) \rightarrow b\ c(b) \in P$, $S \rightarrow b\ c(b)$ is not inserted for usefulness of b, if there exists $S \rightarrow b'\ c(b') \in P$ such that $b' \neq b$ and $d(b') = d(b)$.

Since $d(X) \cap d(Y) = \varnothing$, $d(X) \cap d(Z) = \varnothing$. This leads to $d(X) \cap d(M \cup N) = \varnothing$, because $d(M \cup N) \cap d(Z) = \varnothing$. Hence, G' is not equivalent to G (Lemma 6.15).■

6.4 Comparison to Grammar-Based Mutation Operators

In this section, the grammar-based mutation operators which are defined in [134][18] are discussed and the reasons to avoid using these operators are demonstrated.

- It is possible to realize these operators except for nonterminal duplication as combinations of the event-based mutation operators.
- Mutants generated using these operators generally tend to model multiple missing event or extra event faults.
- Some sets of event-based faults are modeled in different mutants again and again.
- Direct application of these operators except for nonterminal duplication results in RGs, not k-Regs. More critically, nonterminal duplication yields a CFG.

One should note that any set of mutation operators may have issues similar to the above. However, for event-based mutation operators defined in this work, such issues can be avoided quite easily due to their proximity to the event-based fault types.

6.4.1 Nonterminal Replacement

When a nonterminal replacement is performed on a production of a k-Reg, the context of the k-sequence in the tail part is replaced by the context of another k-sequence, but only in this production. Thus, the resulting model is an RG, but it is no longer a k-Reg. Still, one can use the combinations of the event-based mutation operators to realize the nonterminal replacement so that the resulting models are k-Regs (See Table 6.1).

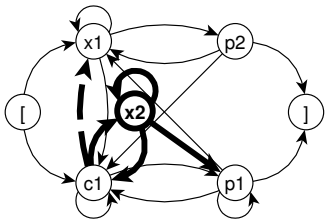
Table 6.1. Nonterminal Replacement in Terms of Event-Based Mutations

Original Production	$S \rightarrow r\ c(r)$	$c(s) \rightarrow r\ c(r)$
Mutated Production	$S \rightarrow r\ Q$	$c(s) \rightarrow r\ Q$
Event-Based Mutations	1. Mark Nonstart: r 2. Insert Terminal: r', where r' is a new k-sequence having the same basis with r 3. Mark Start: r' 4. Insert Sequences: (r', t), for each existing $Q \rightarrow t\ c(t)$ 5. Mark Finish: r', if $Q \rightarrow \varepsilon$ exists	1. Omit Sequence: (s, r) 2. Insert Terminal: r', where r' is a new k-sequence having the same basis with r 3. Insert Sequence: (s, r') 4. Insert Sequence: (r', t), for each existing $Q \rightarrow t\ c(t)$ 5. Mark Finish: r', if $Q \rightarrow \varepsilon$ exists

As demonstrated in Table 6.1, although the change incurred by a nonterminal replacement is very small, the number of missing event or extra event faults modeled by the resulting mutant tends to be relatively large.

$S \rightarrow c1\ c(c1) \mid x1\ c(x1)$
$c(c1) \rightarrow c1\ c(c1)$ ~~$\mid x1\ c(x1)$~~ $\mid p1\ c(p1)$
$\mid x2\ c(x2)$
$c(x1) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid p2\ c(p2)$
$c(p1) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid p1\ c(p1) \mid \varepsilon$
$c(p2) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid \varepsilon$
$c(x2) \rightarrow c1\ c(c1) \mid x2\ c(x2) \mid p1\ c(p1)$

(a) 1-Reg.



(b) Directed graph visualization.

Figure 6.5. A nonterminal replacement mutant of the 1-Reg in Figure 2.2.

Example 6.5 (A Nonterminal Replacement Mutant). Figure 6.5 shows an example nonterminal replacement mutant of the 1-Reg in Figure 2.2 where

nonterminal *c(x1)* in production *c(c1)* → *x1 c(x1)* is replaced by nonterminal *c(c1)*. Note that the same mutant is obtained when nonterminal *c(x1)* is replaced by nonterminal *c(p1)*.

6.4.2 Terminal Replacement

When a terminal replacement is performed on a production of a k-Reg, the context of the replacing k-sequence in the tail of this production is replaced by the context of replaced k-sequence, but only in this production. This results in an RG, not a k-Reg. However, one can use the event-based mutation operators to realize the terminal replacement and assure that the resulting models are k-Regs (See Table 6.2).

Table 6.2. Terminal Replacement in Terms of Event-Based Mutations

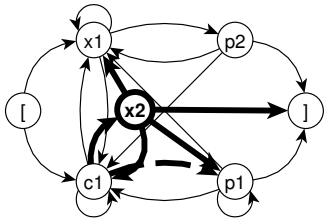
Original Production	S → r c(r)	c(s) → r c(r)
Mutated Production	S → q c(r)	c(s) → q c(r)
Event-Based Mutations	1. Mark Nonstart: r 2. Insert Terminal: q', where q' is a new k-sequence having the same basis with q 3. Mark Start: q' 4. Insert Sequence: (q', t), for each existing c(r) → t c(t) 5. Mark Finish: q', if c(r) → ε exists	1. Omit Sequence: (s, r) 2. Insert Terminal: q', where q' is a new k-sequence having the same basis with q 3. Insert Sequence: (s, q') 4. Insert Sequence: (q', t), for each existing c(r) → t c(t) 5. Mark Finish: q', if c(r) → ε exists

As shown in Table 6.2, a terminal replacement performs a small change. However, the resulting mutant mutant tends to model various missing event or extra event faults at the same time.

Example 6.6 (A Terminal Replacement Mutant). Figure 6.6 demonstrates a mutant of the 1-Reg model in Figure 2.2 which is obtained by performing terminal replacement in production *c(c1)* → *p1 c(p1)*, where *p1* is replaced by *x1*. Note that the 1-Reg models in Figure 6.6 and Figure 6.5 have several common changes. More radically, if terminal replacement is performed on production *c(c1)* → *c1 c(c1)* by replacing *c1* with *x1*, exactly the same mutant in Figure 6.5 is obtained.

S → c1 c(c1) | x1 c(x1)
c(c1) → c1 c(c1) | x1 c(x1) ~~| p1 c(p1)~~
| x2 c(x2)
c(x1) → c1 c(c1) | x1 c(x1) | p2 c(p2)
c(p1) → c1 c(c1) | x1 c(x1) | p1 c(p1) | ε
c(p2) → c1 c(c1) | x1 c(x1) | ε
c(x2) → c1 c(c1) | x1 c(x1) | p1 c(p1) | ε

(a) 1-Reg.



(b) Directed graph visualization.

Figure 6.6. A terminal replacement mutant of the 1-Reg in Figure 2.2.

6.4.3 Nonterminal Deletion

When a nonterminal deletion is performed on a production of a k-Reg, the k-sequence in the tail part is marked as finish, but only in this production. Thus, the resulting model is no longer a k-Reg, but an RG. The nonterminal deletion can be realized in terms of the event-based mutation operators (See Table 6.3). Thus, one can guarantee that the resulting mutants are k-Regs.

Table 6.3. Nonterminal Deletion in Terms of Event-Based Mutations

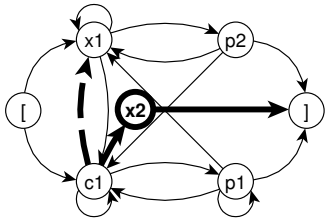
Original Production	S → r c(r)	c(s) → r c(r)
Mutated Production	S → r	c(s) → r
Event-Based Mutations	1. Mark Nonstart: r 2. Insert Terminal: r', where r' is a new k-sequence having the same basis with r 3. Mark Start: r' 4. Mark Finish: r'	1. Omit Sequences: (s, r) 2. Insert Terminal: r', where r' is a new k-sequence having the same basis with r 3. Insert Sequence: (s, r') 4. Mark Finish: r'

Although the change performed by a nonterminal deletion is very small, the number of missing event or extra event faults modeled by the resulting mutant tends to be relatively large as shown in Table 6.3, especially if the other k-sequences in the models become unreachable due to the mutation.

Example 6.7 (A Nonterminal Deletion Mutant). Figure 6.7 demonstrates a nonterminal deletion mutant of the 1-Reg in Figure 2.2, where nonterminal *c(x1)* in production *c(c1) → x1 c(x1)* is deleted. Note that most of the event-based faults modeled by this mutant are already modeled by the mutant in Figure 6.6.

S → c1 c(c1) | x1 c(x1)
c(c1) → c1 c(c1) ~~| x1 c(x1)~~ | p1 c(p1)
| x2 c(x2)
c(x1) → c1 c(c1) | x1 c(x1) | p2 c(p2)
c(p1) → c1 c(c1) | x1 c(x1) | p1 c(p1) | ε
c(p2) → c1 c(c1) | x1 c(x1) | ε
c(x2) → ε

(a) 1-Reg.



(b) Directed graph visualization.

Figure 6.7. A nonterminal deletion mutant of the 1-Reg in Figure 2.2.

6.4.4 Terminal Deletion

When a terminal deletion is performed on a production of a k-Reg, the context of the k-sequence in the tail of this production is included in the context in the head of the production, but only in this production. The resulting mutant is an RG, but not a k-Reg. Nevetheless, the event-based mutation operators can be used to realize the terminal deletion while assuring the resulting models to be k-Regs (See Table 6.4).

Table 6.4. Terminal Deletion in Terms of Event-Based Mutations

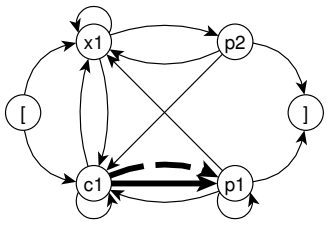
Original Production	S → r c(r)	c(s) → r c(r)
Mutated Production	S → c(r)	c(s) → c(r)
Event-Based Mutations	1. Mark Nonstart: r 2. Mark Start: t, for each existing c(r) → t c(t)	1. Omit Sequence: (s, r) 2. Insert Sequence: (s, t) for each existing c(r) → t c(t) 3. Mark Finish: s, if c(r) → ε exists

As shown in Table 6.4, the change incurred by a terminal deletion is very small. However, the resulting mutant tends to model several extra event faults (and a missing event fault).

Example 6.8 (A Terminal Deletion Mutant). Figure 6.8 demonstrates a terminal deletion mutant of the 1-Reg in model in Figure 2.2 obtained by deleting terminal *p1* in production *c(c1) → p1 c(p1)*. Note that the 1-Reg model in Figure 6.8 is exactly the same as the original 1-Reg model in Figure 2.2.

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

(a) 1-Reg.



(b) Directed graph visualization.

Figure 6.8. A terminal deletion mutant of the 1-Reg in Figure 2.2.

6.4.5 Nonterminal Duplication

When a nonterminal duplication is performed on a production of a k-Reg, the resulting mutant is neither a k-Reg nor an RG; it is a CFG. Hence, in general, it cannot be converted to a k-Reg. Therefore, it is not possible to realize the nonterminal duplication in terms of the event-based mutation operation operators.

Unfortunately, there are several major problems with the use of CFGs. For example, the equivelence is not decidable; in general, one cannot know for sure whether two CFGs are equivalent to each other. Furthermore, one cannot decide whether a given CFG describes a regular language, that is, whether it can be converted to an RG. Such undecidability results for CFGs [100] make them infeasible to use.

Table 6.5 shows how terminal duplication is performed on the productions of a given k-Reg.

Table 6.5. Nonterminal Duplication

Original Production	S → r c(r)	c(s) → r c(r)
Mutated Production	S → r c(r) c(r)	c(s) → r c(r) c(r)

Example 6.9 (A Nonterminal Duplication Mutant). Figure 6.9 demonstrates a nonterminal duplication mutant of the 1-Reg in Figure 2.2, where nonterminal *c(p1)* in production *c(c1)* → *p1 c(p1)* is duplicated.

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

Figure 6.9. A nonterminal duplication mutant of the 1-Reg in Figure 2.2 (a CFG).

6.4.6 Terminal Duplication

When a terminal duplication is performed on a production of a k-Reg, the k-sequence in the tail of this production is repeated before itself to be followed by only itself, but only in this production. This results in an RG, not a k-Reg. However, the terminal duplication can be prformed in terms of the event-based mutation operators so that the resulting models are k-Regs (See Table 6.6).

Table 6.6. Terminal Duplication in Terms of Event-Based Mutations

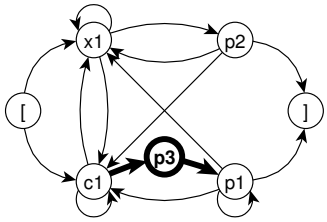
Original Production	S → r c(r)	c(s) → r c(r)
Mutated Production	S → r r c(r)	c(s) → r r c(r)
Event-Based Mutations	1. Mark Nonstart: r 2. Insert Terminal: r', where r' is a new k-sequence having the same basis with r 3. Mark Start: r' 4. Insert Sequence: (r', r) 5. Mark Finish: q', if c(r) → ε exists	1. Omit Sequence: (s, r) 2. Insert Terminal: r', where r' is a new k-sequence having the same basis with r 3. Insert Sequence: (s, r') 4. Insert Sequence: (r', r) 5. Mark Finish: q', if c(r) → ε exists

As implied by Table 6.6, the resulting mutant tends to model various missing event faults and may model an extra event fault, although only a small change is performed by a terminal duplication.

Example 6.10 (A Terminal Duplication Mutant). Figure 6.10 demonstrates a mutant of the 1-Reg model in Figure 2.2 which is obtained by performing terminal duplication in production *c(c1) → p1 c(p1)*, where *p1* is duplicated.

S → c1 c(c1) | x1 c(x1)
c(c1) → c1 c(c1) | x1 c(x1) ~~| p1 c(p1)~~
| p3 c(p3)
c(x1) → c1 c(c1) | x1 c(x1) | p2 c(p2)
c(p1) → c1 c(c1) | x1 c(x1) | p1 c(p1) | ε
c(p2) → c1 c(c1) | x1 c(x1) | ε
c(p3) → p1 c(p1)

(a) 1-Reg.



(b) Directed graph visualization.

Figure 6.10. A terminal duplication mutant of the 1-Reg in Figure 2.2.

7 Mutant Selection for Test Generation

In general, one can create infinitely many mutants modeling multiple missing event or extra event faults. Generation of mutants with relatively large number of faults or with large number of common faults tends to increase the number of mutants, and also the likelihood of a fault making another fault undetectable. Furthermore, it becomes harder to avoid mutants which are equivalent to or a submodel of the original model, or multiple mutants which model the same faults.

Even if mutants which model a small number of faults are generated, it is not a good practice to use all of these mutants for test generation (or MBMT) without considering whether a mutant models a fault which is modeled by another mutant or whether it is equivalent to or submodel of the original model. Such mutants can be used to generate test cases which are not previously generated. However, they increase the total number of test cases significantly and decrease the efficiency of the testing process. Also, these test cases are not guaranteed to contribute to the testing process (other than increasing the number of test cases), because, for example, if a coverage criterion is used, most of the test cases generated from mutants do not improve the intended coverage.

For this reason, this chapter discusses different ways to determine and use mutation operators with certain set of parameters for test generation based on [30][31]. More precisely, different mutant selection strategies are proposed for test generation considering the nature of event-based testing process. The proposed mutant selection strategies have briefly the following properties.

- Each selected mutant models a small number of faults which are located at the mutation points so that one modeled fault does not interfere with another.
- There is no need to compare each mutant to the original model to check for equivalence or to generate distinguishing test cases.
- The generation of equivalent mutants and multiple mutants modeling the same faults are avoided. Thus, correct functioning of the test object with respect to the modeled faults can be checked by exercising different, nonredundant test paths.

- A test case to kill the mutant, that is, a test case to check whether the fault modeled by the mutant exists on a specific path in the test object, can be generated in linear time.

The following assumptions/observations are made on the event-based testing process to propose the mutant selection strategies.

A1. Events in a test case are executed in the given order; therefore, execution of a test case stops when a failure is observed.

A2. The last event of a test case can be any event; a test case needs not to end with a finish event.

Consequently, for a given k-Reg, the following can be stated considering the discussion in Chapter 6.

P1. Missing and extra event faults are limited by considering the k-sequences which 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 are always submodels of the k-Reg. Hence, they do not contain any k-sequence that is not contained in the k-Reg; whereas, the k-Reg may contain k-sequences that are not contained in these mutants. (by 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. Insert sequence mutants are discarded because extra event faults modeled using insert sequence mutants can be modeled using insert terminal mutants. (by Definition 6.5 and Definition 6.6)

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. (By definition [134][18]; also see Section 6.4)

P6. There is no need to continue execution of a negative test case beyond the first faulty sequence. Thus, all negative test cases used in testing process are FCES. (by A1)

In the light of the discussion above, one can perform MBMT by using (k+1)-sequences, faulty 1-sequences and faults (k+1)-sequences as coverage objects or test targets. The original k-Reg can be used to cover (k+1)-sequences for missing event faults (as discussed in Section 5.2), and mark start and insert terminal mutants can be used to cover faulty 1-sequences and faulty (k+1)-sequences for

extra event faults. In this way, one can generate test cases that check the existence of the aforementioned missing and extra event faults on different test paths in the test object. Consequently, only mark start and insert terminal operators are utilized to propose mutant selection strategies and develop test generation methods.

7.1 Mark Start Mutant Selection

Mark start mutant selection strategy can be devised as follows.

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

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

Let G be a useful and deterministic k-Reg. By Theorem 6.1, Theorem 6.2 and Theorem 6.3, 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 where the mutation; that is, the modeled fault is an extra start event fault where e_1 (also $d(e_1)$) is the extra start event for each $Ms(G, e)$.

To discuss the number of mutants generated using the above strategy, let

- $I = \{r \in K |$ there exist no $S \rightarrow x\, c(x) \in P$ such that $d(x_1) = d(r_1)\}$, and
- A be the number of partitions of I such that, in each partition, k-sequences start with 1-sequences having the same basis.

The above strategy relies on marking a single k-sequence from each partition as start. Therefore, the number of generated mutants is given by

$$A \leq |B| \text{ and } A \leq |K| - s$$

where $|K| - s$ is the maximum number of mark start mutans (See Section 6.1.1). Each of these mutants $Ms(G, e)$ can be generated in $O(|P|+|K|) = O(|P|)$ time by checking whether there exists no start k-sequence x such that $d(x_1) = d(e_1)$ and checking whether there exists a previously selected mutation parameter y such that $d(y_1) = d(e_1)$. Also, from each selected mutant, a unique test case that kills it can be generated in $O(1)$ time by simply taking e_1.

The leftout mark start mutants are useful. However, they are either nondeterministic or model previously modeled faults. Some of the nondeterministic mutants do not model any faults at all. In case they do, these

faults are not located at the mutation points; that is, they are not extra start event faults. Therefore, they can be modeled using insert terminal mutants.

Algorithm 7.1 selects mark start mutants using the above strategy. The worst-case runtime complexity is given by $O(|B||P|)$.

- The number of mutants generated is bounded by $|B|$ because each mutant represents a different extra start event fault.
- Each mutant $Ms(G, e)$ can be generated in $O(|P|+|B|) = O(|P|)$ time by checking if there are no start k-sequence x such that $d(x_1) = d(e_1)$ and previously selected mutation parameter y such that $d(y_1) = d(e_1)$, and copying G to modify it.

Algorithm 7.1. Mark Start Mutant Selection

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

Example 7.1 (Mark Start Mutant Selection). Let G be the 1-Reg in Figure 2.2. 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 $Ms(G, p1)$.

7.2 Insert Terminal Mutant Selection

The strategy to select insert terminal mutants is given as follows.

Insert Terminal Mutant Selection. Given a k-Reg $G = (E, B, K, C, S, P)$. For each insert terminal mutant $It(G, e, U, V)$ of G where $U = \{(a, e) | a \in \{a_1, \ldots, a_m\} \subseteq K\}$ and $V = \{(e, b) | b \in \{b_1, \ldots, b_n\} \subseteq K \cup \{e\}\}$, k-sequence e and sets U and V are selected as a mutation parameter if the following conditions hold:

1. $U = \{(a, e)\}$ for some $a \in K$, and, thus, $S \rightarrow e\ c(e)$ is not inserted for usefulness of e.
2. $V = \varnothing$ and $c(e) \rightarrow \varepsilon$ is inserted for usefulness of e.
3. There exists no $c(a) \rightarrow x\ c(x) \in P$ such that $d(x_k) = d(e_k)$.
4. There exists no previously selected mutation parameter $(y, \{(a, y)\}, \{\})$ such that $d(y_k) = d(e_k)$.

Let G be a useful and deterministic k-Reg. By Theorem 6.16, Theorem 6.17 and Theorem 6.18, 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, the modeled fault is an extra event fault where e_k (also $d(e_k)$) is an extra event after k-sequence a for each $It(G, e, U, V)$. Note that each of these mutants models a single fault because the sizes of incoming and outgoing sequence sets are limited.

To discuss the number of mutants generated using the above strategy, let

- $K = \{s_1, s_2, \ldots, s_{|K|}\}$,
- $I(s_i) = \{r |\ r$ is a k-sequence, $r \notin K$, $d(r) \in B^k$ and there exist no $c(s_i) \rightarrow x\ c(x) \in P$ such that $d(x_k) = d(r_k)\}$ for $i = 1, \ldots, |K|$, and
- A_i be the number of partitions of set $I(s_i)$ such that, in each partition, k-sequences end with 1-sequences having the same basis.

Consequently, the number of mutants generated using the above strategy is

$$A_1 + A_2 + \ldots + A_{|K|} \leq |K|\ |B| \leq |K|\ |B|^k,$$

where $|K|\ |B|^k$ is the maximum number of insert sequence mutants generated using one ingoing and no outgoing sequences. Each of these mutants $It(G, e, \{(a, e)\}, \{\})$ can be generated in $O(|P|+|K|) = O(|P|)$ time by checking whether there exists no $c(a) \rightarrow x\ c(x) \in P$ such that $d(x_k) = d(e_k)$, and by checking whether there exists no previously selected mutation parameter $(y, \{(a, y)\}, \{\})$ such that $d(y_k) = d(e_k)$. Also, from each selected mutant, a unique test case that kills the mutant can be generated in $O(|P|)$ time by using breadth-first search to reach $a\ e$.

The excluded insert terminal mutants are useful. However, they are either nondeterministic or model previously modeled faults. Furthermore, nondeterministic mutants may not model any faults at all. In case they do, these faults are not located at the mutation points. Nevertheless, such faults are not discarded because they are modeled using other insert terminal mutants by selecting the mutation parameters differently (or applying the mutation to the proper point in the model).

Algorithm 7.2 generates all insert terminal mutants using the above strategy. The worst-case runtime complexity is given by $O(|K|\ |B|\ |P|)$.

- The number of mutants generated is bounded by $|K|\ |B|$ because each mutant represents a different extra event fault following a k-sequence.
- Each mutant $It(G, e, \{(a, e)\}, \varnothing)$ can be generated in $O(|P|+|B|+k) = O(|P|)$ time by checking whether there are no $c(a) \rightarrow x\ c(x) \in P$ so that $d(x_k) = d(e_k)$ and previously selected mutation parameter $(y, \{(a, y)\}, \varnothing)$ so that $d(y_k) = d(e_k)$, preparing e by copying $a_2 \ldots a_k$ to append b', and copying G to modify it.

Algorithm 7.2. Insert Terminal Mutant Selection

Input: $G = (E, B, K, C, S, P)$ – the input grammar
Output: M – the set of selected insert terminal mutants
$M = \varnothing$, $N = \varnothing$
for each $b \in B$ **do**
  if there is no $S \rightarrow x\ c(x) \in P$ such that $d(x_1) = b$ and
    there is no $y \in N$ such that $d(y_1) = b$ **then**
      Select a k-sequence $e \in K$ such that $d(e_1) = b$
      $G' = G$
      $M = M \cup \{Ms(G', e)\}$
      $N = N \cup \{e\}$
  endif
endfor

Example 7.2 (Insert Terminal Mutant Selection). Let G be the 1-Reg in Figure 2.2. One can only use basis terminal p, because c and x can follow all events. The only elected insert terminal mutant is $It(G, p3, \{(p2,p3\}, \{\})$, because only $p2$ is not followed by a p event.

7.3 Test Generation from Mutants

The following is defined as a countepart to the k-sequence coverage (See Definition 5.5) for generation of negative test cases.

Definition 7.1 (Faulty k-sequence Coverage). Given a 1-Reg $G = (E, B, K, C, S, P)$ and a set of sequences $X \subseteq T_N(G)$. X is said to cover a faulty k-sequence r which is not in G, if r appears in a sequence in X. If X covers all k-sequences not in G, it is said to achieve *faulty k-sequence coverage*.

Note that faulty 1-sequence coverage is different from faulty k-sequence coverage for $k \geq 2$ because the faultiness of an event depends on its preceding and

following events. Faulty 1-sequences actually correspond to faulty start events and, therefore, they should be covered at the beginning of the sequences.

In general, faulty k-sequence coverage is used to reveal extra event faults where an event follows or precedes a (possibly empty) sequence of events (although it should not). However, in the light of the discussion in the beginning of Chapter 7, the faulty k-sequences whose last event is faulty are considered here.

Let G be a 1-Reg and G_k be its corresponding k-Reg. Each mark start mutant of G_k selected using the strategy in Section 7.1 contains a different faulty 1-sequence, and it can be used to reveal an extra start event fault. Furthermore, each insert terminal mutant of G_k selected using the strategy in Section 7.2 contains a different faulty (k+1)-sequence, and it can be used to reveal an extra event faults where an event follows a certain k-sequence. Also, since positive test cases can not cover such sequences, they can not 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 can be generated case for each faulty sequence to obtain a reduced test set (Algorithm 7.3). These generated test cases are used to exercise test paths which are not supposed to exist in the test object and to check whether the test object functions correctly with respect to the considered faults.

Algorithm 7.3. Test Generation to Achieve Faulty (k+1)-sequence Coverage

Input: $G = (E, B, K, C, S, P)$ – the input 1-Reg
k – an integer ≥ 1
Output: X – a set of sequences which achieves single-end faulty *(k+1)-sequence* coverage for G
$X = \varnothing$, $Y = \varnothing$
G_k = *transform* G to its corresponding k-Reg //See Algorithm 5.1
for each $G' = Ms(G_k, e)$ selected using the strategy in Section 7.1 **do**
$X = X \cup \{e_1\}$
endfor
for each $G' = It(G_k, e, \{(a, e)\}, \varnothing)$ selected using the strategy in Section 7.1 **do**
s = generate a sequence ending with e by covering production $c(a) \rightarrow e\ c(e)$ from G'
$X = X \cup T^1(s, k)$ //See Algorithm 5.3
endfor

Let M_{Ms} be the set of all mark start mutants of G_k selected using the strategy in Section 7.1, and M_{It} be the set of all insert terminal mutants of G_k selected using the strategy in Section 7.2. For each mutant, the mutation parameter can be used to identify the fault, because, each mutant models a fault located at the mutation point.

- Using mark start mutants, FCESs of length 1 can be generated to cover faulty 1-sequences. Each mark start mutant $G' = Ms(G, e) \in M_{Ms}$ models a single extra start event fault; to generate a sequence exercising this fault, one needs to cover the production of G' of the form

  $$S \rightarrow e\ c(e)$$

  where e_1 is both the faulty 1-sequence to be covered and the FCES to be generated.
- Using insert terminal mutants, FCESs of length (k+1) or more can be generated to cover faulty (k+1)-sequences. Each insert terminal mutant $G' = It(G, e, \{(a, e)\}, \varnothing) \in M_{It}$ models a single extra event fault; to generate a sequence exercising this fault, one needs to cover the production of G' of the form

  $$c(a) \rightarrow e\ c(e)$$

  where $a\ e_k = a_1 \ldots a_k\ e_k$ is the faulty (k+1)-sequence to be covered.

Since faulty 1-sequence coverage is a special case of (and different from) faulty (k+1)-sequence coverage for $k \geq 1$, when test cases are generated to cover faulty (k+1)-sequences using a k-Reg model, the generation of test cases covering faulty 1-sequences from this k-Reg model is also included.

The worst-case running time complexity of Algorithm 7.3 is given by

$$O((k-1)|P|^{k-1} + |B|\ |P|^k + |K|^k\ |B|\ |P|^{2k}),$$

where

- $O((k-1)|P|^{k-1})$ is the running time complexity of performing $k-1$ consecutive grammar transformations.
- $O(|B|\ |P|^k)$ is the worst-case running time complexity of iterating through all mark start mutants of G_k using the strategy in Section 7.1; a distinguishing test case can be generated from each mark start mutant in $O(1)$ time.
- $O(|K|^k\ |B|\ |P|^k)$ is the worst-case running time complexity of iterating through all insert terminal mutants of G_k using the strategy in Section 7.1; a distinguishing test case can be generated from each insert terminal mutant in $O(|P|^k)$ time.

Example 7.3 (Test Sets Generated Using Algorithm 7.3). When Algorithm 7.3 is executed on the 1-Reg in Figure 2.2 for $k = 1$, one can obtain the following test set.

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

Furthermore, if $k = 3$ is used, the given 1-Reg is transformed twice to obtain the corresponding 3-Reg, the mutants of this 3-Reg is used to obtain test cases. The following is an example test set.

{p1, c1 x1 p2 p3, x1 x1 p2 p3, c1 p1 x1 p2 p3, x1 p2 x1 p2 p3}

As usual, the corresponding basis event is used for each event during test execution.

8 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 [64], to ESG-based MBMT approach (which employs coverage-based test generation from mutants) [40][99], and to 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) [17][47][48][8][11]. Note that the k-Reg-based testing method is not compared to the fault domain-based testing methods using FSMs, because they consider only the faults that can be detected using positive testing (See Section 3.3). Thus, the comparison would be reduced to comparing a coverage-based testing method to a fault domain-based testing method, which is not the subject of this work.

While performing the case studies, the following are carefully considered. (1) The SUCs (or the test objects) 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 Chapter 7 remain valid.

The case studies seek to answer the following questions to demonstrate the improvements of the proposed k-Reg-based MBMT approach.

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 which are not really targeted by the approach?

Q5. How is the test execution trend associated with each approach?

Several metrics exist for generic software processes [62][63]; however, relatively very few are proposed for test processes [75][74]. In this work, the

following metrics are utilized to perform quantitative evaluations and comparisons, and to answer the questions above.

- Number of revealed faults; an indicator of fault detection effectiveness
- Test generation time and total number of executed events; indicators of cost effectiveness or testing costs
- Fault detection rate; an indicator of fault detection efficieny (or test efficiency)
- Number of faults revealed as events are executed; an indicator of test execution trends demonstrating how fault detection effectiveness changes as tests are executed

Furthermore, total number of mutants is used to demonstrate the effectiveness of the mutant selection strategies of the proposed approach when compared to the other MBMT approaches.

8.1 Experimental Design and Parameters

To make appropriate comparisons, *test targets* [22] 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.

8.1.1 k-Reg

The *k-Reg* approach makes use of only the corresponding k-Reg in order to generate test cases. More specifically, positive test cases (or, more precisely, CESs) achieving (k+1)-sequence coverage and negative test cases (or, more precisely, FCESs) achieving faulty (k+1)-sequence coverage are generated from the given k-Reg and mutants of this k-Reg using Algorithm 5.4 and Algorithm 7.3, respectively. Note that by choosing *k* values appropriately, the testing cost can be adjusted to make the approach scalable for larger applications.

8.1.2 Mixed k-Reg (M-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. Positive test cases (or, more precisely, CESs) are generated from the given (k+1)-Reg for achieving (k+2)-sequence coverage (Algorithm 5.4), and negative test cases (or, more precisely, FCESs) are generated from the mutants of the given k-Reg for achieving faulty (k+1)-sequence coverage (Algorithm 7.3). Therefore, it can also be considered as a half way between the *k-Reg* and the *(k+1)-Reg* approaches.

8.1.3 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 case is bounded by *maxlen*.

In *Random(k+1,maxlen)*, positive test cases (or, more precisely, start sequences (SS)) achieving (k+1)-sequence coverage and negative test cases (or, more precisely, FCESs) achieving faulty (k+1)-sequence coverage are generated from the given 1-Reg, adapting the approach defined in [22] as follows.

- Random SSs are sampled as positive test cases. To do this, a random test case length *len* where $1 \leq len \leq maxlen$ is selected first. Later, a random SS of the selected length is sampled using the given 1-Reg. If this sequence covers a remaining (k+1)-sequence, it is added to the test set and the covered (k+1)-sequence is removed from the remaining (k+1)-sequences. This process is repeated until no more (k+1)-sequences remain to cover.
- Random FCESs as negative test cases. First, a random prefix length *len* where $0 \leq len \leq maxlen$ is selected. Later, a random prefix of the selected length, which is either an empty sequence or a SS, is selected using the given 1-Reg. If possible, the selected prefix is completed with a related faulty event. If the resulting FCES sequence covers a remaining faulty (k+1)-sequence, it is added to the test set and the covered faulty (k+1)-sequence is removed from the remaining faulty (k+1)-sequences. This process is repeated until no more faulty (k+1)-sequences remain to cover.

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 [20].

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

Researches suggest that random testing performs better than a large class of testing strategies [20][21]. Also, unlike most other random testing adaptations which do not use any information about the program or the specification [53], the approach adapted in this work uses information on the test targets [22] derived using k-Reg models. Thus, this adaptation can be considered to quite competitive.

8.1.4 ESG(k+1)

The *ESG(k+1)* represent the ESG-based MBMT counterpart of the *k-Reg* and the *M-k-Reg* approaches. An ESG is equivalent to a 1-Reg model without a set of basis events and a set of 1-sequences. In *ESG(k+1)*, no test targets are defined; instead, the given ESG model is mutated using mutation operators and test cases are generated from each mutant by achieving (k+1)-sequence coverage [40][99]. No mutant selection or morphology variation is carried out. Hence, mutants which are sub models of or equivalent to the original model, and multiple mutants modeling the same faults are also used in test generation.

Since too many mutants can be generated (See Table 8.1), the test generation time tends to be too long and the size of the generated test tends to be quite large. For this reason, 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 *ESG(k+1)*, a corresponding size is selected as the executable size of the test set generated using either *k-Reg* or *M-k-Reg*.
- ESG mutants are randomly selected and test cases are generated from these mutants as long as the executable test set size does not exceed the selected size.

ESG(k+1,k) is used to refer to the ESG-based counterpart of *k-Reg*, and *ESG(k+1,M-k)* is used to refer to that of *M-k-Reg*. Thus, the modification is used to balance ESG-based approach separately with *k-Reg* and *M-k-Reg* in terms of the test execution effort.

8.1.5 MK (Mutate and Kill)

The *MK* represents the mutate-and-kill-based (MK-based) MBMT approach which is based on generating discriminating test cases [17][47][48][8][11] 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 left out, multiple mutants modeling the same faults are used in test generation.

Similar to *ESG(k+1)*, due to the large number of possible mutants (See Table 8.1), *MK* approach is modified to limit the executable size of the generated test set.

- The executable sizes of the test sets generated using *k-Reg* and *M-k-Reg* approaches are selected as the limits.

- 1-Reg mutants are randomly selected and test cases are generated from these mutants as long as the executable test set size does not exceed the corresponding limit.

The modification is used to balance MK-based approach separately with *k-Reg* and *M-k-Reg* in terms of the test execution effort; *MK(k)* is used to refer to the MK-based counterpart of *k-Reg*, and *MK(M-k)* is used to refer to that of *M-k-Reg*.

(a) SFH. (b) ShearBar.

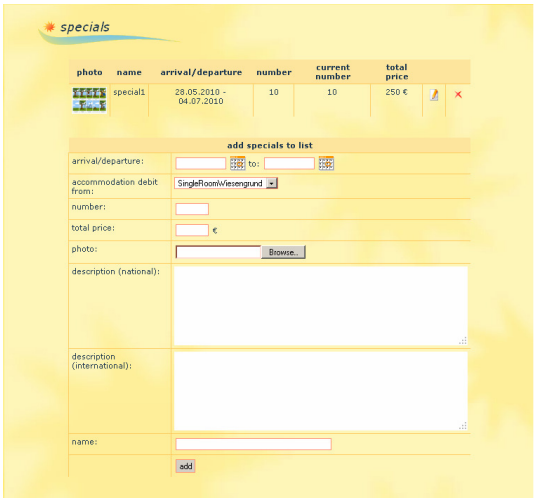
Figure 8.1. SFH of CLAAS.

8.2 Systems Under Consideration (SUCs)

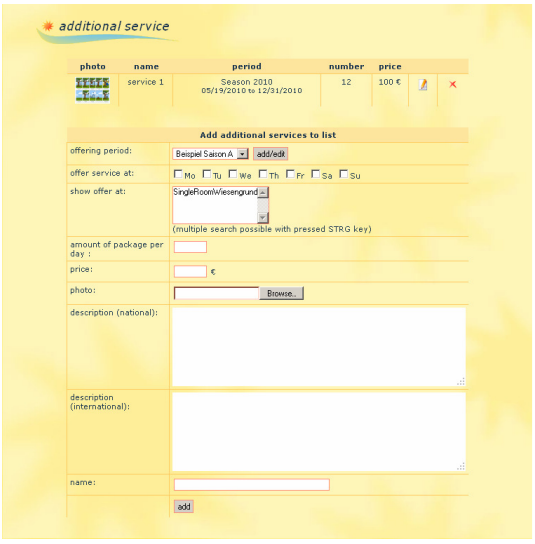
For the case studies, three nontrivial SUCs are selected from two commercial systems: (1) SFH (Self-propelled Forage Harvester) of CLAAS (http://www.claas.com) and (2) ISELTA (Isik's System for Enterprise-Level Web Centric Tourist Applications) of Isik Touristik (http://www.isik.de).

SFH (Figure 8.1a) is a farm implement that harvests forage plants to make silage; it picks up the plants like grass or corn, chops them into small pieces and loads them into a wagon in one working process. 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 (Figure 8.1b) as the first case study. The `ShearBar` is controlled by signals coming from various external sources; its function is very critical for safety and financial reasons.

(a) ISELTA



(a)Specials.



(b) Additionals.

Figure 8.2. ISELTA of Isik Touristik.

ISELTA (Figure 8.2a) is a commercial web portal for marketing tourist services. It enables travel and tourism enterprises, such as hotel owners and agencies, to create their own individual search and service masks. These masks can be embedded in the existing homepage of the enterprises as an interface between the customers and the system. 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` (Figure 8.2b) and `Additionals` (Figure 8.2c).

Table 8.1. k-Reg Models

ShearBar					
k	**k-Reg Productions**	**k-Reg Mutants**	**Total Mutants**	**k-sequences**	**Faulty k-sequences**
1	422	32364	31058682	314	103
2	558	40653	-	395	32261
3	698	52077	-	506	40574
4	856	-	-	626	51998

Specials					
k	**k-Reg Productions**	**k-Reg Mutants**	**Total Mutants**	**k-sequences**	**Faulty k-sequences**
1	429	755	737370	90	12
2	2191	3378	-	427	743
3	11205	17732	-	2184	3367
4	58019	-	-	11171	17221

Additonals					
k	**k-Reg Productions**	**k-Reg Mutants**	**Total Mutants**	**k-sequences**	**Faulty k-sequences**
1	513	804	813285	93	13
2	2984	4189	-	511	791
3	17271	24454	-	2977	4177
4	100869	-	-	17236	24442

8.3 Models of the SUCs

A 1-Reg model is created for each SUC from the system specification, and k-Reg models for $k \geq 2$ are obtained using k-Reg transformation (Algorithm 5.1). The size and complexity of k-Reg models for each SUC are included in Table 8.1 to assure that they are not trivial (Refer to Appendix A for the 1-Reg models).

In addition, Table 8.1 gives the numbers of mutants that are selected using the k-Reg-based approach and that can be generated using the existing event-based mutation operators or event-base MBMT approaches, which employ no mutant selection strategies. Since other MBMT approaches do not vary model morphology, mutant numbers are counted using the initial system models.

As can be seen in Table 8.1, the total numbers of mutants selected using different k-Reg models are ~0.40%, ~2.97% and ~3.62% of the total numbers of mutants that can be generated. This shows the effectiveness of the mutant

selection strategies discussed in Section 7. Also, it demonstrates why the ESG-based and the MK-based MBMT approaches are modified using a size parameter as described in Section 8.1.4 and Section 8.1.5.

Note that computing the exact numbers of equivalent mutants and the exact number of multiple mutants modeling the same faults is not feasible, because mutants need to be compared to the original model and to the other mutants. The proposed strategies avoid such mutants without distinguishing them from each other as described in Section 7.

Furthermore, Table 8.1 demonstrates that the relation between k and the number of test targets is different for each SUC.

8.4 Fault Seeding

Due to large number of possible mutants for each SUC, a fixed number of event-based faults are randomly generated and seeded [135][90][171] to compare the testing processes in a realistic manner while gaining insight into test execution. Such an insight is not provided by mutation analysis since the faults are considered separately.

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 because the length of and the number of k-sequences that need to be covered increase and a stronger coverage is generally required to systematically uncover the related faults.

k-Reg, *M-k-Reg, Random(k+1)*, *ESG(k+1,k)* and *ESG(k+1,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 relevant for real-world faults. There is evidence supporting the fact that a test set that detects more model model-based mutants also detects more code-based mutants [10] and that a test set that detects more code-based mutants also detects more real-world faults [19]. Thus, one can conclude that a test set that detects more model-based mutants also detects more real-world faults.

8.5 Test Generation and Execution Results

Lower bounds for *maxlen* are selected to guarantee that the intended test targets can be covered and in reasonable time, and the upper bounds are selected to avoid excessive test generation and execution times. Thus, *maxlen* is limited to

- *60,63,67,70* for `ShearBar` and
- *20,30,40,50* for `Specials` and `Additionals`.

Also, to collect precise data on test execution process, test cases are executed in the following way.

- Each test case 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 test case reveals no faults and runs until completion, it is not executed again.
- This process continues until all test cases are executed to completion.

Appendix B contains the test generation and execution results.

- Table B.1, Table B.2 and Table B.3, and Table B.4, Table B.5 and Table B.6 present summarized data on test set generation and test execution processes.
- Table B.7, Table B.8 and Table B.9 outline the data on the number of revealed faults (See Section 8.4 for *m*).
- Figure B.1, Figure B.2 and Figure B.3 demonstrate how the revealed number of faults changes with respect to the number of events executed.

8.6 Interpretation of Results

Questions Q1 - Q5, which are posed at the beginning of Chapter 8, can be now answered in order.

8.6.1 Q1: Fault Detection Effectiveness

The revealed fault numbers are rounded to the nearest integers for comparison of fault detection effectiveness, and Table 8.2, Table 8.3 and Table 8.4 are constructed by rewriting the data for *k-Reg* and *M-k-Reg* from Table B.4, Table B.5 and Table B.6 with respect to *Random(k+1), ESG(k+1)* and *MK*, respectively.

Table 8.2. Fault Detection Effectiveness w.r.t. Random(k+1)

		Faults Revealed w.r.t. Random(k+1)		
		k=1	**k=2**	**k=3**
ShearBar	**k-Reg**	5.58% (10) fewer	0.55% (1) fewer	same
	M-k-Reg	0.43% (1) fewer	0.55% (1) more	0.52% (1) more
Specials	**k-Reg**	24.41% (23) fewer	9.80% (13) fewer	3.79% (7) fewer
	M-k-Reg	0.79% (1) more	8.69% (12) more	3.19% (5) more
Additionals	**k-Reg**	25.07% (22) fewer	11.80% (15) fewer	6.53% (11) fewer
	M-k-Reg	6.05% (5) more	7.54% (10) more	2.76% (5) more

Table 8.3. Fault Detection Effectiveness w.r.t. ESG(k+1)

		Faults Revealed w.r.t. ESG(k+1)		
		k=1	**k=2**	**k=3**
ShearBar	**k-Reg**	23.13% (31) more	38.93% (51) more	40.58% (56) more
	M-k-Reg	28.89% (39) more	40.46% (53) more	41.30% (57) more
Specials	**k-Reg**	56.52% (26) more	64.86% (48) more	71.13% (69) more
	M-k-Reg	77.78% (42) more	77.11% (64) more	66.36% (71) more
Additionals	**k-Reg**	71.05% (27) more	78.13% (50) more	73.12% (68) more
	M-k-Reg	84.00% (42) more	78.21% (61) more	60.91% (67) more

Table 8.4. Fault Detection Effectiveness w.r.t. MK

		Faults Revealed w.r.t. MK		
		k=1	**k=2**	**k=3**
ShearBar	**k-Reg**	60.19% (62) more	76.70% (79) more	79.63% (86) more
	M-k-Reg	68.93% (71) more	78.64% (81) more	80.56% (87) more
Specials	**k-Reg**	80.00% (32) more	134.62% (70) more	100.00% (83) more
	M-k-Reg	123.26% (53) more	113.04% (78) more	81.63% (80) more
Additionals	**k-Reg**	71.05% (27) more	100.00% (57) more	98.77% (80) more
	M-k-Reg	124.39% (51) more	107.46% (72) more	77.00% (77) more

Table 8.2 demonstrates that, in general, *k-Reg* reveals fewer faults than *Random(k+1)*, up to ~25.07%. However, *M-k-Reg* almost always performs better than *Random(k+1)* by revealing up to ~8.69% more faults. Furthermore, Table 8.3 shows that both *k-Reg* and *M-k-Reg* always reveal more faults than their corresponding *ESG(k+1)* counterparts, respectively, up to ~78.13% and ~84.00%, and Table 8.4 shows that they also always reveal more faults than their corresponding *MK* counterparts, respectively, up to ~134.62% and ~124.39%.

When the change in the number of faults revealed is considered with respect to *k* (Table B.4, Table B.5 and Table B.6), *k-Reg* shows the overall fastest increasing trend for each case study, being up to ~1.06 times faster than *Random(k+1)* and it is followed by *M-k-Reg*, being up to ~27.5% faster. Furthermore, *ESG(k+1)* and *MK* may sometimes even show decreasing trends. When the increasing trends are considered, *k-Reg* and *M-k-Reg* are up to ~91.30% and ~75.86% faster than *ESG(k+1,k)* and *ESG(k+1,M-k)*, respectively, and, they are up to ~3.2 times and ~1.2 times faster than *MK(k)* and *MK(M-k)*, respectively.

8.6.2 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. . 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 [170][32][33]. Thus, Table 8.5 is constructed using the data in Table B.1, Table B.2 and Table B.3 by rounding test generation times appropriately and calculating how much fewer events are executed with respect to random testing. The numbers of events executed by k-Reg-based approaches are not discussed with respect to the ESG-based and the MK-based approaches because the approaches are balanced in terms of the test execution effort (See Section 8.1.4 and Section 8.1.5).

Table 8.5 shows the effects of linear-time test generation from the mutants in the k-Reg-based testing approach. In general, test generation times are quite smaller for *k-Reg* and *M-k-Reg* when compared to *Random(k+1)*, *ESG(k+1,k)*, *ESG(k+1,M-k)*, *MK(k)* and *MK(M-k)*, up to ~99.99%. However, in some cases, test generation times for *M-k-Reg* are greater when compared to *MK(M-k)*, up to ~4.5 times, because M-k-Reg uses the corresponding (k+1)-Reg (instead of the k-Reg) for positive test case generation. Also, *k-Reg* and *M-k-Reg* require, respectively, up to ~70.65% and ~33.62% less test execution efforts than *Random(k+1)*.

In addition, Table B.1, Table B.2 and Table B.3 suggest that as *k* increases, test generation time increases,respectively, up to ~99.99% to ~99.98% less for *k-Reg* and *M-k-Reg* when compared to *Random(k+1)*. Furthermore, it increases, respectively, up to ~99.92% and ~99.90% less when compared to *ESG(k+1,k)* and *ESG(k+1,M-k)*. Also, although the increase is generally up to ~99.96% less for *k-Reg* and *M-k-Reg* respectively when compared to *MK(k)* and *MK(M-k)*, it is sometimes greater for *M-k-Reg* when compared to *MK(M-k)*, up to 12.4 times. As for the change in test execution effort with increasing *k*, *k-Reg* and *M-k-Reg* show,

respectively, up to ~67.09% and ~17.26% less increase when compared to *Random(k+1)*.

Table 8.5. Test Generation and Test Execution Costs

		Test Generation Time			Fewer Events Executed w.r.t. Random(k+1)		
		k=1	**k=2**	**k=3**	**k=1**	**k=2**	**k=3**
ShearBar	**k-Reg**	6 s	12 s	19 s	13.89%	13.62%	12.91%
	M-k-Reg	6 s	12 s	20 s	13.73%	13.57%	12.82%
	Random(k+1)	13.8 h	15.2 h	26.6 h	-	-	-
	ESG(k+1,k)	24.9 h	22.2 h	24.6 h	-	-	-
	ESG(k+1,M-k)	24.9 h	22.3 h	24.5 h	-	-	-
	MK(k)	11.9 h	15.8 h	20.7 h	-	-	-
	MK(M-k)	12.0 h	15.8 h	20.8 h	-	-	-
Specials	**k-Reg**	1 s	4 s	58 s	68.94%	66.46%	62.77%
	M-k-Reg	3 s	49 s	1.09 h	33.62%	20.73%	14.81%
	Random(k+1)	12.5 m	3.4 h	46.1 h	-	-	-
	ESG(k+1,k)	10 s	44 s	22 m	-	-	-
	ESG(k+1,M-k)	24 s	148 s	1.4 h	-	-	-
	MK(k)	12 s	44 s	272 s	-	-	-
	MK(M-k)	18 s	99 s	12.0 m	-	-	-
Additionals	**k-Reg**	1 s	6 s	170 s	70.65%	67.74%	63.82%
	M-k-Reg	5 s	150 s	3.6 h	26.55%	11.34%	5.13%
	Random(k+1)	1.1 h	20.0 h	12 d	-	-	-
	ESG(k+1,k)	13 s	67 s	52.8 m	-	-	-
	ESG(k+1,M-k)	50 s	5.7 m	5.5 h	-	-	-
	MK(k)	9 s	55 s	5.9 m	-	-	-
	MK(M-k)	23 s	131 s	18.1 m	-	-	-

* s=seconds, m=minutes, h=hours, d=days

8.6.3 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 8.6.2, FDR is also formulated as the inverse of *cost per detected fault (CPF)*; that is, *FDR = 1 / CPF*.

Using the data in Table B.4, Table B.5 and Table B.6, the differences in FDRs with respect to *Random(k+1)*, *ESG(k+1)* and *MK* are given in Table 8.6, Table 8.7 and Table 8.8, respectively.

According to Table 8.6, 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 *ESG(k+1,k)* and *ESG(k+1,M-k)*, respectively, up to ~79.37% and ~81.29% (as shown in Table 8.7), and than *MK(k)* and *MK(M-k)*, respectively, up to ~153.21% and ~144.12% (as shown in Table 8.8).

Table 8.6. Fault Detection Efficiency w.r.t Random(k+1)

		Fault Detection Rate w.r.t. Random(k+1)		
		k=1	**k=2**	**k=3**
ShearBar	**k-Reg**	9.61% higher	15.08% higher	14.77% higher
	M-k-Reg	15.37% higher	16.29% higher	15.24% higher
Specials	**k-Reg**	130.20% higher	154.53% higher	143.69% higher
	M-k-Reg	43.62% higher	29.77% higher	14.19% higher
Additionals	**k-Reg**	140.66% higher	158.06% higher	143.27% higher
	M-k-Reg	36.11% higher	14.48% higher	1.99% higher

Table 8.7. Fault Detection Efficiency w.r.t ESG(k+1)

		Fault Detection Rate w.r.t. ESG(k+1)		
		k=1	**k=2**	**k=3**
ShearBar	**k-Reg**	23.13% higher	38.85% higher	40.46% higher
	M-k-Reg	28.78% higher	40.31% higher	41.15% higher
Specials	**k-Reg**	54.49% higher	69.73% higher	72.56% higher
	M-k-Reg	70.49% higher	81.29% higher	67.67% higher
Additionals	**k-Reg**	78.88% higher	79.31% higher	79.37% higher
	M-k-Reg	80.51% higher	78.37% higher	61.16% higher

Table 8.8. Fault Detection Efficiency w.r.t MK

		Fault Detection Rate w.r.t. MK		
		k=1	**k=2**	**k=3**
ShearBar	**k-Reg**	59.92% higher	76.76% higher	79.26% higher
	M-k-Reg	68.87% higher	78.61% higher	80.65% higher
Specials	**k-Reg**	153.21% higher	147.39% higher	102.04% higher
	M-k-Reg	144.12% higher	117.98% higher	82.09% higher
Additionals	**k-Reg**	105.70% higher	108.35% higher	99.81% higher
	M-k-Reg	123.28% higher	107.38% higher	77.65% higher

As for the change in FDR as *k* increases (Table B.4, Table B.5 and Table B.6), all approaches show decreasing trends. *k-Reg* shows, respectively, up to

~162.35%, ~79.29% and ~165.74% faster decreasing trend than *Random(k+1)*, *ESG(k+1,k)* and *MK(k)*, and *M-k-Reg* shows, respectively, up to ~49.59%, ~85.45% and ~155.57% faster decreasing trend than *Random(k+1)*, *ESG(k+1,M-k)* and *MK(M-k)*. Nevertheless, as mentioned above, FDRs of *k-Reg* and *M-k-Reg* always remain greater than *Random(k+1)*, *ESG(k+1)* and *MK*.

8.6.4 Q4. Effectiveness of detecting Non-targeted Faults

As already mentioned in Section 8.4, *k-Reg*, *M-k-Reg* and *Random(k+1)* approaches aim to detect faults modeled using a k-Reg. Table B.7, Table B.8 and Table B.9 suggest 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 ESG-based MBMT and MK-based MBMT approaches since each of these approaches uses a single fixed model.

Table 8.9. Fault Detection Effectiveness w.r.t. Random(k+1) - Detailed

		Percentage of Faults Revealed w.r.t. Random(k+1) for m=1,2,3,4		
		k=1	**k=2**	**k=3**
ShearBar	**k-Reg**	0.00, -0.53, -11.84, -12.00 (Overall: -7.59)	0.00, 0.00, **1.34**, -3.60 (Overall: -0.63)	0.00, 0.00, 0.00, -100.00 (Overall: -0.06)
	M-k-Reg	0.00, -0.53, -1.76, **1.33** (Overall: -0.36)	0.00, 0.00, **3.64**, -1.07 (Overall: **0.87**)	0.00, 0.00, 0.00, **2.09** (Overall: **0.64**)
Specials	**k-Reg**	0.00, -45.98, -46.86, -100.00 (Overall: -51.23)	0.00, 0.00, -23.27, -66.16 (Overall: -15.57)	0.00, 0.00, 0.00, -26.38 (Overall: -6.27)
	M-k-Reg	0.00, **8.04**, **6.29**, -52.38 (Overall: **1.97**)	0.00, 0.00, **40.67**, **18.44** (Overall: **13.74**)	0.00, 0.00, 0.00, **22.70** (Overall: **4.53**)
Additionals	**k-Reg**	0.00, -46.79, -77.25, -100.00 (Overall: -59.17)	0.00, 0.00, -36.47, -72.63 (Overall: -18.83)	0.00, 0.00, 0.00, -46.04 (Overall: -9.08)
	M-k-Reg	0.00, **30.98**, -8.99, -43.18 (Overall: **14.32**)	0.00, 0.00, **58.81**, -8.76 (Overall: **12.87**)	0.00, 0.00, 0.00, **20.38** (Overall: **4.02**)

* A positive value means more; a negative value means fewer.
\+ Overall values are calculated considering only the nontargeted faults.

Using the data in Table B.7, Table B.8 and Table B.9, Table 8.9, Table 8.10 and Table 8.11 are 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)*, *ESG(k+1)* and *MK* for *m=1,2,3,4* (See Section 8.4 for *m*).

Table 8.9 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, Table 8.10 shows that *k-Reg* and *M-k-Reg* are overall up to ~68.42% and ~133.33% more effective than *ESG(k+1,k)* and *ESG(k+1,M-k)*, respectively, and Table 8.11 show that they are also overall up to ~94.59% and ~180.00% more effective than *MK(k)* and *MK(M-k)*, respectively.

Table 8.10. Fault Detection Effectiveness w.r.t. ESG(k+1) - Detailed

		Percentage of Faults Revealed w.r.t. ESG(k+1) for m=1,2,3,4		
		k=1	**k=2**	**k=3**
ShearBar	**k-Reg**	**47.06**, **30.56**, **16.67**, -2.94 (Overall: **15.00**)	**51.52**, **38.89**, **37.50**, **26.67** (Overall: **38.95**)	**51.52**, **38.89**, **42.86**, **29.41** (Overall: **39.81**)
	M-k-Reg	**47.06**, **27.03**, **30.00**, **11.76** (Overall: **22.77**)	**51.52**, **38.89**, **40.63**, **30.00** (Overall: **41.05**)	**51.52**, **38.89**, **42.86**, **32.35** (Overall: **40.78**)
Specials	**k-Reg**	**72.41**, **60.00**, **20.00**, -100.00 (Overall: **29.41**)	**78.57**, **100.00**, **20.00**, -33.33 (Overall: **46.94**)	**75.00**, **81.48**, **85.19**, **20.00** (Overall: **65.71**)
	M-k-Reg	**61.29**, **128.57**, **71.43**, 0.00 (Overall: **100.00**)	**47.06**, **92.31**, **94.12**, **133.33** (Overall: **70.18**)	**40.00**, **63.33**, **85.19**, **100.00** (Overall: **60.00**)
Additionals	**k-Reg**	**85.19**, **44.44**, 0.00, **n/a** (Overall: **36.36**)	**88.46**, **92.31**, **20.00**, **50.00** (Overall: **68.42**)	**77.78**, **85.19**, **100.00**, -7.14 (Overall: **63.24**)
	M-k-Reg	**56.25**, **113.33**, **166.67**, **n/a** (Overall: **133.33**)	**63.33**, **56.25**, **130.77**, **233.33** (Overall: **93.48**)	**50.00**, **51.52**, **85.19**, **61.11** (Overall: **53.01**)

* A positive value means more; a negative value means fewer.
\+ Overall values are calculated considering only the nontargeted faults.

Table 8.11. Fault Detection Effectiveness w.r.t. MK - Detailed

		Percentage of Faults Revealed w.r.t. MK for m=1,2,3,4		
		k=1	**k=2**	**k=3**
ShearBar	**k-Reg**	**78.57**, **88.00**, **34.62**, **37.50** (Overall: **53.33**)	**78.57**, **100.00**, **69.23**, **58.33** (Overall: **69.23**)	**78.57**, **85.19**, **78.57**, **76.00** (Overall: **80.00**)
	M-k-Reg	**78.57**, **100.00**, **69.23**, **58.33** (Overall: **65.33**)	**78.57**, **100.00**, **73.08**, **62.50** (Overall: **71.79**)	**78.57**, **85.19**, **78.57**, **80.00** (Overall: **81.25**)
Specials	**k-Reg**	**100.00**, **60.00**, **200.00**, -100.00 (Overall: **46.67**)	**72.41**, **233.33**, **350.00**, 0.00 (Overall: **94.59**)	**36.11**, **75.00**, **316.67**, **157.14** (Overall: **63.38**)
	M-k-Reg	**92.31**, **190.91**, **500.00**, -50.00 (Overall: **170.59**)	**47.06**, **127.27**, **312.50**, **180.00** (Overall: **106.38**)	**22.50**, **44.12**, **212.50**, **275.00** (Overall: **56.10**)
Additionals	**k-Reg**	**100.00**, **18.18**, 0.00, 0.00 (Overall: **15.38**)	**58.06**, **150.00**, **300.00**, 0.00 (Overall: **72.97**)	**37.14**, **61.29**, **525.00**, **85.71** (Overall: **52.05**)
	M-k-Reg	**92.31**, **166.67**, **166.67**, **n/a** (Overall: **180.00**)	**48.48**, **100.00**, **650.00**, **100.33** (Overall: **111.90**)	**26.32**, **38.39**, **233.33**, **163.64** (Overall: **49.41**)

* A positive value means more; a negative value means fewer.
\+ Overall values are calculated considering only the nontargeted faults.

8.6.5 Q5. Test Execution Trends

`ShearBar.` The overall execution trends of *k-Reg* and *M-k-Reg* for `ShearBar` (Figure B.1) are very similar to each other, especially as *k* increases. All the approaches reveal faults very quickly at the beginning of the test execution. Half of all the revealed faults are discovered by performing ~0.5% of the test execution for *k-Reg* and *M-k-Reg*, ~1% for *Random(k+1)*, ~0.3% for *ESG(k+1)* and ~0.5% for *MK*.

For *Random(k+1)*, *ESG(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 significant and sudden increase. The first point resides between ~5% and ~7% of the test execution and the second point resides around ~95%.

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

revealed faults for the last ~5% of the execution. In addition, *k-Reg* and *M-k-Reg* achieves better FDRs than *ESG(k+1)* and *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, respectively, up to ~2.48% and ~8.07% more faults 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 ~40.58% and ~41.30% more faults than their *ESG(k+1)* counterparts, and, 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)*, *ESG(k+1)* and *MK* test execution curves for `Specials` (Figure B.2) are quite different from each other, except for the fact that *k-Reg*, *ESG(k+1,k)* and *ESG(k+1,M-k)* show similar trends for the first up to ~46% of the *k-Reg* test execution, *ESG(k+1,M-k)* is an extension of *ESG(k+1,k)* and *MK(M-k)* is an extension of *MK(k)*.

For *Random(k+1)*, the rate of change in FDR decreases as the test execution proceeds. After ~40% of the test execution is completed, a sudden increase in the rate of change in FDR occurs. This also happens for *ESG(k+1,k)* and *ESG(k+1,M-k)*, respectively, after around ~18% and ~8% of the test execution. Such increases are more frequent for *k-Reg* and *M-k-Reg*, but the most significant ones happen, 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)*, less apparent for *ESG(k+1,k)* and *ESG(k+1,M-k)*, 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, except for the fact that *M-3-Reg* is ~9.28% less effective than *ESG(4,M-3)*, *k-Reg* and *M-k-Reg* are more effective than *ESG(k+1,k)* and *ESG(k+1,M-k)*, respectively, up to ~71.13% and ~4.35%, and they 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 and *M-k-Reg* is more effective than *Random(k+1)* except until the end stages where they sometimes reach similar values. In addition, *ESG(k+1)* is more or equally effective as all the other approaches for the first ~27.84% (for *k=1*), ~46.44% (for *k=2*) and ~38.13% (for *k=3*) of the test execution, and *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%, ~77.78% and ~123.26% more effective than *Random(k+1)*, *ESG(k+1,M-k)* 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 less than *ESG(k+1,M-k)* for the first ~19.50% (for *k=1*), ~17.19% (for *k=2*) and ~56.13% (for *k=3*) of the test execution; and it is similar to *ESG(k+1,M-k)* from ~19.50% to ~46.73% (for *k=1*), from ~17.19% to ~52.75% (for *k=2*), and from ~56.13% to ~76.36% (for *k=3*) of the test execution. Also, 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)*.

`Additionals.` The shapes of test execution curves for `Additionals` (Figure B.3) are relatively similar to those of `Specials`. *k-Reg*, *ESG(k+1,k)* and *ESG(k+1,M-k)* show similar trends for the first up to ~55.38% of the *k-Reg* test execution, *ESG(k+1,M-k)* is an extension of *ESG(k+1,k)*, and *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 *ESG(k+1,k)*, *ESG(k+1,M-k)*, *M(k)* and *MK(M-k)*, even if such increases happen, they are not very significant; whereas, for *M-k-Reg* and *k-Reg*, they are more frequent and sudden, and they becomes 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 *ESG(k+1,k), ESG(k+1,M-k)*, *MK(k)* and *MK(M-k)*, respectively, up to ~78.13%, ~62.50%, ~100.00% and ~19.30%, except for the fact that *M-3-Reg* is ~7.58% less effective than *ESG(4,M-3)*. 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, *ESG(k+1)* is more or equally effective as all the other approaches for the first ~34.58% (for *k=1*), ~27.26% (for *k=2*) and ~55.38% (for *k=3*) of the test execution, and *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%, ~84.00% and ~124.39% more effective than *Random(k+1)*, *ESG(K+1,M-k)* 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 *ESG(k+1,M-k)* for the first ~28.27% (for *k=1*), ~28.84% (for *k=2*) and ~60.75% (for *k=3*) of the test execution, and, for *k=3*, *M-k-Reg* and *ESG(k+1,M-k)* show quite similar trends from ~60.75% to ~83.62%. Furthermore, 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 rest.

8.7 A Brief Summary of Results

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

- for `ShearBar`,
  - ~2% fewer faults while making ~13% less test execution effort, that is, ~13% more faults per executed event, when compared to the random testing approach,
  - ~34% more faults and ~34% more faults per executed event, when compared to the ESG-based MBMT approach by balancing the test execution effort, and
  - ~72% more faults and ~72% more faults per executed event, when compared to the MK-based MBMT approach by balancing the test execution effort;
- for `Specials`,
  - ~13% fewer faults while making ~66% less test execution effort, that is, ~143% more faults per executed event, when compared to the random testing approach,
  - ~64% more faults and ~66% more faults per executed event, when compared to the ESG-based MBMT approach by balancing the test execution effort, and

  - ~105% more faults while and ~134% more faults per executed event, when compared to the MK-based MBMT approach by balancing the test execution effort; and

- for `Additionals`,
  - ~14% fewer faults while making ~67% less test execution effort, that is, ~147% more faults per executed event, when compared to the random testing approach,
  - ~74% more faults and ~79% more faults per executed event, when compared to the ESG-based MBMT approach by balancing the test execution effort, and
  - ~90% more faults and ~105% more faults per executed event, when compared to the MK-based MBMT approach by balancing the test execution effort.

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

- for `ShearBar`,
  - the same number of faults while making ~13% less test execution effort, that is, ~16% more faults per executed event, when compared to the random testing approach,
  - ~37% more faults and ~37% more faults per executed event, when compared to the ESG-based MBMT approach by balancing the test execution effort, and
  - ~76% more faults and ~76% more faults per executed event, when compared to the MK-based MBMT approach by balancing the test execution effort;
- for `Specials`,
  - ~4% more faults while making ~23% less test execution effort, that is, ~29% more faults per executed event, when compared to the random testing approach,
  - ~74% more faults and ~73% more faults per executed event, when compared to the ESG-based MBMT approach by balancing the test execution effort, and
  - ~106% more faults and ~115% more faults per executed event, when compared to the MK-based MBMT approach by balancing the test execution effort; and
- for `Additionals`,
  - ~5% more faults while making ~14% less test execution effort, that is, ~18% more faults per executed event, when compared to the random testing approach,

- ~74% more faults and ~73% more faults per executed event, when compared to the ESG-based MBMT approach by balancing the test execution effort, and
- ~103% more faults and ~103% more faults per executed event, when compared to the MK-based MBMT approach by balancing the test execution effort.

Using the number of executed events as an indicator of the test execution effort and assuming that the average effort required executing each event is approximately the same, the above data suggest that k-Reg-based testing reveals either most of the faults with generally considerable less effort or even more faults with yet significantly less effort, when compared to random testing. However, it always remains superior to random testing in terms of efficiency which is quantified by fault detection rate. Also, when compared to ESG-based and MK-based testing by balancing the test execution efforts, k-Reg-based testing is always more effective and efficient.

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 `Additionals`) of the numbers of mutants required by MBMT approaches with no mutant selection strategies.

As mentioned in Section 8.2, 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 $\sim 1.32e^{0.1497k}$ as fast as `Specials` (See Table 8.1 for trends). The results suggest that the differences between *k-Reg/M-k-Reg* and *Random(k+1)*, *k-Reg/M-k-Reg* and *ESG(k+1)* and *k-Reg/M-k-Reg* and *MK* are relatively less apparent for `ShearBar` when compared to `Specials` and `Additionals`.

Thus, the relation between the number of test targets and *k* seems to affect the difference between the results observed using different approaches.

8.8 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. [169]

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, three diametrically different SUCs representing different typical situations are selected and used. 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 [22] is used and 30 test sets [20] are generated for each *Random(k+1, maxlen)* and 4 different *maxlen* values are selected. Therefore, 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 [53].

In addition, the ESG-based and the MK-based MBMT approaches are modified by balancing them against the k-Reg-based testing approach in terms of the test execution effort as described in Section 8.1.4 and Section 8.1.5. This is likely to reduce the fault detection effectiveness of the approaches for the sake of completing the case studies in a feasible time as discussed in Section 8.1.4, Section 8.1.5 and Section 8.3.

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 8.4) 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, there is also an upper bound to avoid relatively high test generation and execution efforts. 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 execution 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 8.6.1), 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 8.6.3), 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 B.1, Table B.2 and Table B.3.

Part III

Further Aspects and Concluding Remarks

9 Further Perspectives

In this chapter, further steps are taken along three different directions to broaden the research perspectives. These three directions are

- application of the approach for modeling, testing and analysis of system vulnerabilities (based on [34]),
- application of MBMT using models which are not purely event-based but have stronger expressiveness (based on [36][37] and [161]), and
- extension of the event-based models in different ways (based on [35]).

For the sake of simplicity and brevity, the discussion in this chapter is mostly semi-formal.

9.1 Modeling, Testing and Analysis of System Vulnerabilities

When observing an interactive human-machine system, desirable and undesirable behaviors, or events, are differentiated depending on the expectations of the user concerning the system behavior. Desirable events include those related to global critical system properties such as reliability, safety, and security. Any deviation from the expected behavior defines an undesirable state; the fact that the system can be transferred into such a state might be viewed as a *vulnerability* of the system. A vulnerability is often accompanied by *threats*. Therefore, a complementary view of the desirable system behavior is necessary for a holistic modeling, analysis, and testing of the system.

This work uses the term “vulnerability” to refer to any behavior where the violation of the requirements of a system attribute, such as safety or security, may lead to a significant penalty in terms of cost, damage, or harm. In the case of *safety*, the threat originates from within the system due to potential failures and its spillover effects causing potentially extensive damage to its environment. In the face of such failures, the environment could be a helpless, passive victim. The goal of the system design in this case is to prevent faults that could potentially

lead to such failures or, in worst cases, to mitigate the consequences of run-time failures should they ever occur. In the case of *security*, the system is exposed to external threats originating from the environment, causing losses to the owner of the system. In this case, the environment, typically the user, maybe unauthorized, can be malicious or deliberately aggressive. The goal of system design then is to ensure that it can protect the system itself against such malicious attacks. Although, in this work, the outlined approach is used to test safety aspects, it is applicable to other vulnerability attributes like security, etc.

In the face of such vulnerabilities, testing forms an important part of the system development process in revealing and eliminating faults in the system. In addition, it continues to play an essential role during the maintenance phase. Due to the substantial costs involved in testing, both testability and the choice of tests to be conducted become important design considerations. Due to the conflicting demands of minimizing the extent of tests and maximizing the coverage of faults, it is therefore critically important to follow a systematic approach to identifying the test sets that focus on safety, as well as tests that address specific safety requirements [117][157].

With the above in view, this work proposes an approach where the test design can progress hand in hand with the design process, paying particular attention to safety. It is based on a formal (rule-based, graphic, or algebraic) representation of the system and its environment, potentially including the user. User actions and system events in the representation, referred to here as “events” for simplicity, are ordered according to the threats posed by the resulting system states. This ordering is an integral aspect of the finite state representation, making it possible to directly identify the risks associated with each and every functionally desirable, and undesirable, event relative to one another. Tests that target safety requirements are devised by examining possible traces (sequences) of these events exhibiting particular risk patterns. These patterns are represented by regular expressions (REs). The undesirable events in them represent human error and system failures, while the desirable events include, in addition to functional ones, various recovery measures to be undertaken following undesirable events.

The approach is model-based. It enables an incremental refinement of the model and specification, which at the beginning may be rough and rudimentary, or even nonexistent. However, the approach can also be deployed in implementation-oriented analysis and test in a refined format, for example, using the implementation (source code) as a concise description of the system under consideration (SUC), that is, as the ultimate specification, and its control flow diagram as a state transition diagram (STD) (see also [82][145]). To sum up, the approach can be used not only for requirements analysis and validation before implementation, but also for analysis and testing of an existing implementation,

detecting input/output faults, erroneous internal states, etc., at low levels of abstraction.

A broader objective of this research is to develop a single framework for dealing with different system vulnerability attributes, carrying risks of different nature and degrees of severity, that is, safety and security, broadly in a similar manner, while capturing their fundamental differences by an appropriate characterization of the risks involved. This work is a formal, detailed and, hopefully, an intuitive introduction to the approach.

An early version of the proposed approach was introduced in [44][45]. The work in [66][42] addresses different vulnerability aspects like user-friendliness and safety that have different implications due to potential human error. Furthermore, the work also discusses certain concepts related to test generation based on event sequence graphs (ESGs). This work uses a special form of regular grammars (RGs) (more precisely, k-Regs - Section 4) to discuss and extend the concepts and outline concrete algorithms, while using directed graph visualizations (or ESGs) and REs as alternative respresentations. Furthermore, it focuses on the testability of event-based systems against vulnerabilities (based on, for example, safety), and it demonstrates test design that targets vulnerability aspects as part of an integrated system development process.

The remainder of this section is organized as follows. Section 9.1.1 introduces different models of the SUC that are equivalent to each other and to a finite state automaton (FSA). Section 9.1.2 discusses modeling system functions and vulnerability threats, and model-based testing and analysis. The concept of risk ordering, introduced in Section 9.1.3 in relation to attributes such as safety, is another fundamental concept of the approach and is applicable to other vulnerability system attributes as well. In Section 9.1.4, a real-life example is given to illustrate the use of risk ordering as the means of: (a) modeling safety aspects and (b) designing tests for verifying these properties.

9.1.1 Models of the SUC

This work focuses on event-based modeling using k-Regs for representation of the user and system behavior. The set of input/output signals (or events) E of a SUC can be partitioned into two subsets E_{env} and E_{sys} such that

$$E = E_{env} \cup E_{sys} \text{ and } E_{env} \cap E_{sys} = \varnothing, \quad (1)$$

where E_{env} is the set of *environmental events* (for example, user inputs) while E_{sys} is the set of *system signals* (for example, responses). The distinction between the sets E_{env} and E_{sys} is important because the events in the latter are controllable from within the system, whereas the events in the former are not subject to such control.

As already discussed in Section 4, k-Regs are equivalent to RGs. Therefore, they are also comparable with finite state automata (FSA) or Myhill graphs [130] that are used as computation schemes [102], or as *syntax diagrams*, for example, as used in [105][162] to define the syntax of Pascal; see also e*vent sequence* concept as introduced in [111]. The difference between the Myhill graphs and k-Regs is that the symbols, which label the nodes of directed graph visualizations, are interpreted not merely as symbols and metasymbols of a language, but as operations, or even a sequence of operations, of an event set.

k-Regs are also comparable to regular expressions (REs) [150]. However, REs are more of a declarative nature. This work uses REs for describing the patterns of interactivity between the system and its environment, and for identifying system states if required.

A *regular expression (RE) R* over alphabet (of basis events) *d(E)* is inductively defined as follows (letting *L(R)* be the language defined by RE *R*):

- **Base case:** Let $\varnothing$ be the empty language, ε be the empty string and *a* be any symbol in the alphabet, then
  - The constants $\varnothing$ and ε are REs denoting the languages $L(\varnothing) = \varnothing$ and $L(\varepsilon) = \{\varepsilon\}$ respectively.
  - *a* is an RE denoting the language $L(a) = \{a\}$.
- **Inductive case:** If *R* and *P* are two REs, denoting the languages *L(R)* and *L(P)* respectively, then
  - $R+P$, the *union* of *R* and *P*, is an RE denoting the language $L(R+P) = L(R) \cup L(P)$.
  - *R.P*, the *concatenation* of *R* and *P*, is an RE denoting the language $L(R.P) = L(R).L(P)$.
  - R^*, the *closure* of *R*, is an RE denoting the language $L(R^*) = L(R)^*$ and $R^+ = R.R^*$.
  - *(R)*, is an RE denoting the language $L((R)) = L(R)$.

Intuitively, an RE can be assumed to be a sequence of symbols *a, b, c, ...* of an alphabet that can be connected by operations

- sequence (“.” (usually omitted), for example, “*ab*” means “*b* follows *a*”),
- selection (“+,” for example, “*a*+*b*” means “*a* or *b*”), and
- iteration (“*,” *Kleene’s Star Operation*, for example,
  - “a^*” means “zero or more occurrences of *a*”;
  - “a^+” means “at least one occurrence of *a*”).

For example, RE

$$R = (ab(a+c)^+)^*$$

indicates that *a* is followed by *b*, leading to *ab*, which is followed by at least one occurrence of either *a* or *c*. The entire sequence can be repeated an arbitrary number of times. Examples of the generated sequences are: *aba, abc, abaaba, abaabc* but also ε for 0 (zero) occurrence in the sequence.

9.1.2 Behavioral Patterns of the SUC

System functions, as well as the threats to a chosen system vulnerability attribute, may each be described using two disjoint subsets of strings,

- one belonging to the language *d(L(M))* and
- another *not* belonging to *d(L(M))*, respectively,

where $M = (E, B, K, C, P, S)$ is a k-Reg.

Legal state transitions are brought about by *desirable* events, leading to symbol sequences belonging to *d(L(M))* and specifying system functions. *Illegal* transitions represent the *undesirable* events, leading to faulty symbol sequences not belonging to *d(L(M))*, signifying breaches to vulnerabilities.

Let *F* to denote the *system functions* and *V* the *vulnerability threats* such that′

$$F \subseteq d(L(M)) \text{ and } V \subseteq d(L(M)) \tag{2}$$

where *d(L(M))′* is the complement of *d(L(M))*. For testing of event-based systems satisfying the assumptions in Section 7, mark start and insert terminal mutants discussed in Section 7.1 and Section 7.2 can be used for formulation of *V* with some slight changes.

Let M_{Ms} be the set of all marks start mutants of *M* selected using the strategy defined in Section 7.1 such that the following are additionally performed on each $Ms(M, e) \in M_{Ms}$.

- *e* is marked as finish.
- All finish terminals except *e* are marked as nonfinish.
- All terminals from which *e* is not reachable are omitted.

Furthermore, let M_{It} be the set of all insert terminal mutants of *M* selected using the strategy defined in Section 7.2, such that the following are additionally performed on each $It(M, e, \{(a, e)\}, \{\}) \in M_{It}$.

- All finish terminals except *e* are marked as nonfinish.
- All terminals from which *e* is not reachable are omitted.

Then vulnerability threats can be formulated as

$$V \subseteq (d(L(M_{Ms})) \cup d(L(M_{It}))). \tag{3}$$

It is important to note that several vulnerability attributes may simultaneously apply to a given application. In this case, *F* will remain the same in the study of

every vulnerability attribute, but V will vary from one vulnerability attribute to another. To avoid mismatching, the relevant threats to each attribute *att* will be identified as V_{att}.

Depending on the chosen system vulnerability attribute, the strings corresponding to vulnerability threats can be grouped in accordance with their length n. This assumes that all threats can be unequivocally identified by patterns of n consecutive symbols, that is, strings, of E. It is obvious that the grammar M can be utilized to test whether the system functions are fulfilled, and/or vulnerability threats occur by generating CESs and FCESs, respectively (See Section 5.2 and Section 7.3 for more details).

9.1.3 Ordering the Vulnerability Risks, Countermeasures

The vulnerability threats constitute only a part of the specification of system vulnerability. Such threats are often related to the system state. When a representation based on purely events is used, it becomes necessary to refer to states indirectly in terms of a subset of the words in $L(M)$, for example, using REs. However, contexts (or nonterminals) of the grammar can also be utilized to refer to states. Note that a single context corresponds to a single state whereas a single state may correspond to multiple contexts.

Thus, for a given $M = (E, B, K, C, P, S)$, a *vulnerability risk ordering relation* $\sqsubseteq$ is defined on $C \times C$ as

$$\sqsubseteq = \{(s_1, s_2) |\ s_1, s_2 \in C \text{ and } rl(s_1) \leq rl(s_2)\} \subseteq C \times C \tag{4}$$

where $rl(s)$ is the risk level associated with state s. In other words, given two states s_1 and s_2, $s_1 \sqsubseteq s_2$ is true if and only if risk level of s_1 with respect to the chosen system vulnerability is known to be less than or equal to the risk level of s_2 [131]. In this context, *risk level* quantifies the "*degree of the **un**acceptability*" of an event on the grounds of hazardousness, that is, exposure to breaches of safety.

The risk ordering relation $\sqsubseteq$ is intended as a guide to decision making upon the detection of a threat, whether internal or external, and on how to react to it. The required response to breaches of vulnerability needs to be specified in terms of a *defense matrix* D, which is a partial function from $C \times V$ to C. The defense matrix utilizes the risk ordering relation to revert the system state from its current one to a less, or the least, risky state. In this sense, D is defined as

$$\begin{gathered} D: C \times V \rightarrow C \text{ where} \\ \forall s_1, s_2, v \quad (s_1, v) \in domain(D) \text{ and if } D(s_1, v) = s_2 \text{ then } s_2 \sqsubseteq s_1. \end{gathered} \tag{5}$$

The above definition expresses the requirement that, should it encounter the vulnerability v in the state s_1, the system must be brought down to a state s_2 which is of a lower risk level than s_1. The means by which this is brought about is called

an *exception handler*, or a defensive action, which is an appropriate enforced sequence of events. If x is a defense action appropriate for the scenario implicit in (5) above, then $D(s_1, x) = s_2$. The actual definition of the defense matrix and the appropriate set X of *exception handlers* is the responsibility of a domain expert specializing in the risks to a given vulnerability.

In order to ease the construction of the defense matrix, it is possible to make use of risk graphs that partially associate risk levels of functional and vulnerability states.

Given $M = (E, B, K, C, P, S)$, and M_{Ms} and M_{It} as defined in Section 9.1.2, The *risk graph of M* is a tuple $(C_h, \sqsubseteq_h)$, where C_h is the set of *holistic states (or contexts)* and $\sqsubseteq_h$ is the *holistic risk ordering relation* defined as follows.

$$\begin{aligned} C_h &= C \cup \{c(e) | \ e \text{ is a finish terminal in some } G \in (M_{Ms} \cup M_{It})\} \text{ and} \\ \sqsubseteq_h &= \{(s_1, s_2) | \ s_1, s_2 \in C_h \text{ and } rl(s_1) \leq rl(s_2)\} \subseteq C_h \times C_h. \end{aligned} \tag{6}$$

By making use of the risk graph, one can build and use the defense matrix more easily and determine more efficient exception handlers. Of course, while creating the risk graph, the defense matrix and the exception handlers, one should always consider the underlying semantics of the SUC.

Finally, the *model* of an application *defended* against vulnerabilities is defined as

$$M_d = (M, V, \sqsubseteq, D, X), \tag{7}$$

where $M = (E, B, K, C, P, S)$ is a k-Reg, V is the set of vulnerability threats, $\sqsubseteq$ is the risk ordering relation, D is the defense matrix and X is the set of exception handlers.

A specific benefit of risk ordering in the above framework is that it allows a more systematic approach to selection of test cases by focusing on (one or more) particular vulnerability attributes.

9.1.4 Case Study: Railway Crossing

This case study illustrates the use of the approach in the area of safety critical systems, using an example that considers a railway level crossing (Figure 9.1).

9.1.4.1 SUC

Railway crossings, found across minor roads outside towns, normally consist of a pair of gates and two traffic lights: red and green, and also a railway signaling system to control the train movement in the proximity of the crossing, though the latter is ignored here as a simplification. Note that in this model the human is a part of the system environment, for example, as driver, gate controller, etc. Our

holistic approach enables the consideration of the driver's expected, that is, correct, as well as the faulty behavior. Despite its simplicity, the example is sufficiently expressive for our purpose. Note, however, that our discussion is based on an ordinary familiarity of the application and, therefore, our representation may not be quite accurate from a specialist's point of view.

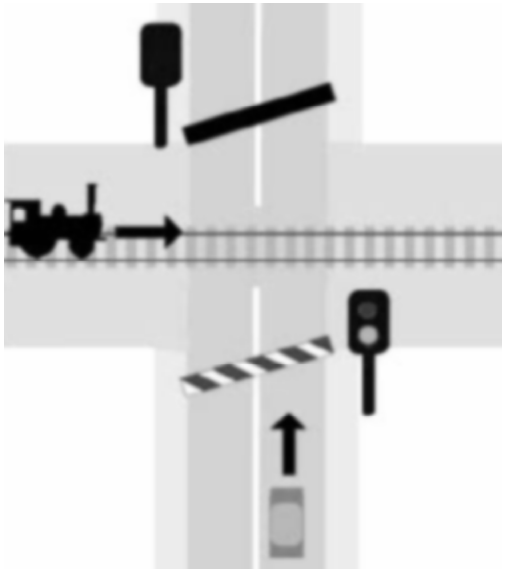


Figure 9.1. Simplified railway crossing.

A 1-Reg model of such a crossing is shown in Figure 9.2. The set of input signals (or signal 1-sequences) E is partitioned into the subsets

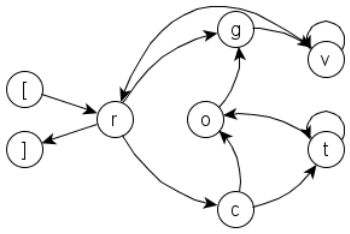
- $E_{sys} = \{r, g, c, o\}$ as *system signals* and
- $E_{env} = \{v, t\}$ as *environmental events* as detected by a system that monitors the crossing.

Here r denotes the event of turning traffic signals to red, g the turning of traffic signals to green, c the closing the gate barring vehicle traffic, as well as other road users, from using the crossing, o the opening the gate allowing vehicle traffic through, t a train passing the crossing, and v for a vehicle using the crossing. These events bring about hazardous states posing different risks to road users and rail users alike. The nature of these hazards varies from state to state, some posing greater threats than others. For example, compared to the safest possible state $c(r)$ (traffic lights being red), the state $c(o)$ (an opened gate) carries a greater risk since the road users are now free to cross the junction, exposing themselves to danger from a passing-by train. Likewise, the state $c(t)$ (a train

actually crossing the junction) poses a greater risk than the state *c(c)* (a closed gate), since the latter includes also cases when there is no train within the crossing.

$S \rightarrow r\ c(r)$
$c(r) \rightarrow g\ c(g) \mid c\ c(c) \mid \varepsilon$
$c(g) \rightarrow v\ c(v)$
$c(c) \rightarrow o\ c(o) \mid t\ c(t)$
$c(o) \rightarrow g\ c(g)$
$c(v) \rightarrow r\ c(r) \mid v\ c(v)$
$c(t) \rightarrow o\ c(o) \mid t\ c(t)$

(a) 1-Reg.



(b) Directed graph visualization.

Figure 9.2. A 1-Reg model of a railway level crossing.

Figure 9.2 also indicates the relative risk levels brought about by the occurrence of the respective events. In the diagram, the states posing greater threats to the users of the system are placed horizontally to the right of those posing relatively lower risks. Note also that, as a simplification, the above representation does not include any means to control the movement of trains. It is assumes that the system is initialized with a sequence of signals *rc*.

9.1.4.2 System Functions and Vulnerability Threats

As implied by the productions of the 1-Reg in Figure 9.2, the 2-sequences in this example are

$$rg,\ rc,\ gv,\ co,\ ct,\ og,\ vr,\ vv,\ to,\ tt. \tag{8}$$

Also, the complete event sequences (CESs) in any complete cycle of system operation can be represented by the following RE

$$(rgv^+)^*r+((rgv^+)^*rct^*ogv^+)^*r = ((rgv^+)^*(\varepsilon + rct^*ogv^+))^*r, \tag{9}$$

which describes the same behavior as the 1-Reg in in Figure 9.2. The difference in the two descriptions lies in the fact that REs are of a declarative nature whereas k-Regs are of an imperative nature. Thus, sometimes it is beneficial to use REs for inline or brief descriptions.

The faulty 2-sequences are in this case

$$\begin{gathered} rr,\ ro,\ rv,\ rt,\ gr,\ gg,\ go,\ gc,\ gt,\ or,\ oo,\ oc,\ ov, \\ ot,\ cr,\ cg,\ cc,\ cv,\ vg,\ vo,\ vc,\ vt,\ tr,\ tg,\ tc,\ tv, \end{gathered} \tag{10}$$

which can be obtained using the insert terminal mutants. Note that mark start mutants are ignored for this example, because it does not make sense to perform a faulty event without properly initializing the system by performing *r c* (See Section 9.1.4.1).

The 1-Reg in Figure 9.2 (or the RE in (9)) describes the *system function F*, while faulty 2-sequences given above are the system *vulnerability threats V* posed at the junctures corresponding to any matching sub-sequences among event sequences that can be generated by the 1-Reg presented in Figure 9.2, for example,

$$(rgv^+)^*r. \tag{11}$$

Each faulty 2-sequence in (10) represents the leading pair of signals of an emerging faulty behavioral pattern, with the first event being an acceptable one and the second one an unacceptable one. Should the first event of any of the faulty 2-sequence, such as *rv* in (10), happen to match the last event in any of the event sequences that can be generated by such a subexpression, such as $(rgv^+)^*r$ in (11), then the corresponding pair of the event sequence and the faulty 2-sequence, such as

$$(rgv^+)^*rv, \tag{12}$$

which describes, or signifies the occurrence of, a specific form of a faulty behavioral pattern.

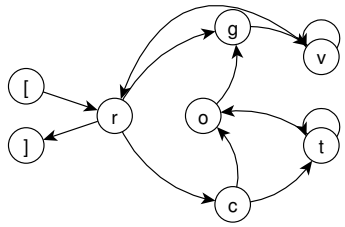
Of course, such sequences can easily be generated by creating a mutant for each faulty 2-sequenceas described in Section 9.1.2, and by selecting words from the languages described by these mutants. Intuitively, this corresponds to the concatenation of the corresponding pairs of event sequences and faulty 2-sequences in the appropriate manner.

Note that it is easy to represent all event sequences ending with terminal *r* (as partially intended in (11) above) using a 1-Reg. Since derivations of the form $S \Rightarrow^* xr\ c(r)\ (x \in K^*)$ generate all such sequencess, it makes sense to simply refer to the set of all such sequences using context *c(r)*. Furthermore, the required CFESs can be formulated by mutants in an easier and a more precise way. For example, an insert terminal mutant of the 1-Reg in Figure 9.2 is shown in Figure 9.3; it contains all forms of faulty behavior patterns induced by the use of faulty 2-sequence *rv*.

Naturally, the words in the language described by such a mutant do not belong to the language described by the original 1-Reg. More precisely, a insert terminal mutant selected as discussed in Section 9.1.2 describes all possible FCESs ending with a specific faulty 2-sequence. Furthermore, one can also use unique contexts of such mutants for representation of vulnerability states.

S → r c(r)
c(r) → g c(g) |c c(c)~~+ε~~ | **v2 c(v2)**
c(g) → v c(v)
c(c) → o c(o) | t c(t)
c(o) → g c(g)
c(v) → r c(r) | v c(v)
c(t) → o c(o) | t c(t)
c(v2) → ε

(a) Insert terminal mutant.



(b) Directed graph visualization.

Figure 9.3. An insert terminal mutant of the 1-Reg in Figure 9.2.

Table 9.1 presents the vulnerabilities relevant to the model given in Figure 9.2. In spite of its simplicity, the interpretations of the conjunctions of the appropriate pairs (event sequence, faulty 2-sequence) demonstrate the effectiveness of the approach in revealing the safety-critical cases. For completeness, and the sake of clarity, to represent event sequences, both contexts and REs that lead to these contexts in the1-Reg are included.

9.1.4.3 Risk Graph and Defense Mechanism

A graph of the form given in Figure 9.4 may be more informative about the relative risk levels. Each node in this *risk graph* is a context that represents any state belonging to the complete state space which includes the functional states (contexts of the 1-Reg model) and the vulnerability states (additional contexts of the mutant models). Each context can be used to signify a state unambiguously; however, a state may correspond to multiple contexts. A directed arc running from a node s_1 to another node s_2 in Figure 9.4 suggests that to $s_1 \sqsubseteq s_2$; that is, the risks posed by s_2 is known to be not lower than the risks posed by s_1 (the risks posed by s_2 can be at the same level as, or exceed, the risks by s_1). Note that the use of upward running arcs in Figure 9.4 is to signify that the state lying above (in the vertical direction) poses a greater risk than the one lying below.

As mentioned above, arcs and nodes drawn in solid lines refer to the normal functional behavior and functional states, while those in dashed lines refer to undesired behavior and vulnerability states. To avoid drawing a spacious graph, some vulnerability states are merged in Figure 9.4; that is, each dashed node in the risk graph corresponds to a subset of vulnerability states. Therefore, the dashed states are not named explicitly. Instead, REs are given to identify them. Furthermore, as indicated by Table 9.1, there can be some FCESs that cause no effective change in the state, for example, in issuing a signal to close the gate when it is already closed. Such transitions have been termed as *futile transitions* in Figure 9.4 and are not considered any further.

Table 9.1. Level Crossing Vulnerabilities, Threat Levels, and Possible Defense Actions

Event Sequence (Column 1)	**Faulty 2-sequence (Column 2)**	**Interpretation (Column 3)**	**Comment (Column 4)**	**Defense Action (Column 5)**
$((rgv^+)^* + (rgv^+)^* rct^* ogv^+))^* r$ $c(r)$	ro	Gate opens while lights are set to red (No effective state change is possible except immediately after initialization when the gate was closed).	Ignored	–
	rt	A train arrives prematurely.	Danger	rc
	rv	Vehicle traffic passes through red lights.	Danger	*
$(rgv^+)^* rc$ $c(c)$	cr	Lights to revert to red, though already red.	Ignored	–
	cv	Vehicle traffic is attempting to cross the closed gate and the red lights.	Danger	*
	cg	Lights turn green from red while the gate is closed.	Danger	*
$(rgv^+)^* rct^+$ $c(t)$	tr	Lights to revert to red while already in red.	Ignored	–
	tc	Gates to close while already closed.	Ignored	–
	tv	Vehicle traffic crosses as trains pass.	Potential accident	None
	tg	Lights turn green as trains pass.	Danger	
$(rgv^+)^* rct^* o$ $c(o)$	or	Lights to revert to red while already in red.	Ignored	–
	oc	Gates to close while already closed.	Ignored	–
	ot	A train arrives after the gate opened.	Danger	rc
	ov	Vehicle traffic crosses as soon as the gate opened but before the lights change to green.	Danger	*
$(rgv^+)^* rct^* og + rg$ $c(g)$	go	Gates to open though already opened.	Ignored	–
	gc	Gates to close after the lights turn green.	Annoyance	
	gt	A train arrives soon after the lights turn green.	Danger	rc
$(rgv^+)^* rct^* ogv^+ + rgv^+$ $c(v)$	vo	Gates to open though already opened.	Ignored	–
	vc	Gates to close while vehicle traffic moving.	Danger	vr
	vt	A train arrives amidst vehicle traffic.	Potential accident	rc
	vg	Lights to turn green though already green.	Ignored	–

* Any defense action is outside the scope of the current model due to lack of features for controlling train movements.

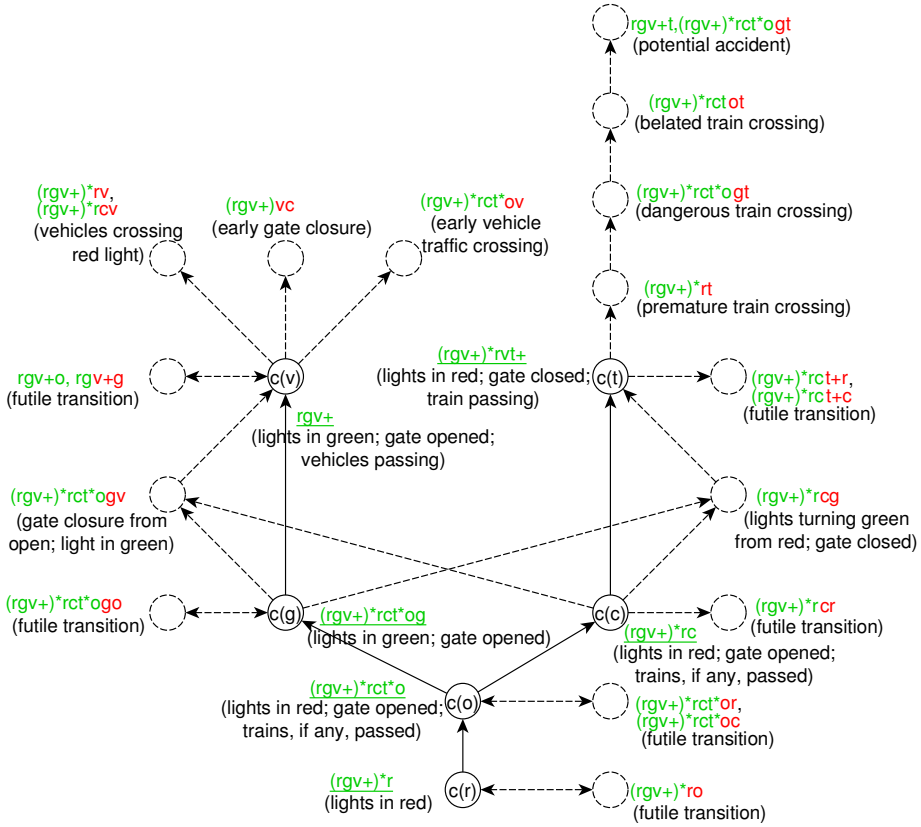


Figure 9.4. Risk graph of the railway crossing, covering both the functional and the vulnerability states.

Having identified potential vulnerabilities of safety concerns, it is possible to provide measures that counteract them. This is intended by the concept of *defense matrix* and *exception handlers* introduced in Section 9.1.3. In this connection, an attempt has been made in Table 9.1 to propose the defensive actions that may be taken, although due to the limited scope of our model these actions address only partially the potential vulnerabilities. This is because all defensive actions at the disposal of the current model are limited to closing the gate or turning the traffic lights to red, thus affecting only the vehicle traffic. A richer model with features for modeling signaling mechanisms would have allowed means to address other vulnerabilities, namely, those that can be avoided or mitigated by controlling the train movements. In a complete Table 9.1 in these respects, the event sequences listed under Column 5 would be equivalent to the set of the exception handlers X,

while Columns 1, 2, and 5 would amount to a definition of the required defense matrix implicitly, provided that the data in these columns satisfies the condition in (5). Note that Column 5 lists the exception handlers, and Columns 1 and 2 give the domain of the defense matrix (function); that is, a context *c(a)* in Column 1 represents set of all possible sequences of the form *xa* such that $S \Rightarrow^* xa\ c(a)$ $(x \in E^*)$ and an *ab* in Column 2 is a faulty 2-sequence. Furthermore, the concatenation of Columns 1, 2, and 5 in the appropriate manner (that is, by dropping common signals as appropriate) leads to the context representing the safe state aimed by the defense matrix as a result of invoking the corresponding exception handler [116].

9.1.4.4 Testing Safety Issues

The testing can now be worked out as defined in Section 5.2 and Section 7.3; the CES and the FCES can be systematically constructed ans used to test the system for desirable and undesirable behaviors.

As for the railway level crossing, actual testing of the application in real life for safety issues cannot be undertaken; firstly, it places human life at risk, secondly, it is impractical on the grounds of costs and, thirdly, it is unnecessary. It is quite hard to run the system with all the test inputs and observe what happens. As an example, the test input (12) represents the event that the vehicle traffic passes through the red lights, which cannot be realized as a real-life experiment. Furthermore, in order to generate a complete test case, a meaningfully reactive controlling system is needed, which is outside the scope of our current model, given the 1-Reg in Figure 9.2. Nevertheless, even this simple approach is useful in that it makes such dangerous behaviors explicit (visible) and highlights the reactions required of the controlling system in response to such inputs. Thus, it is evident that 1-Reg model in Figure 9.2 can be used to simulate all potential test scenarios. All what is required is the proper use of the concepts defined in Sections 9.1.1, 9.1.2 and 9.1.3 by following the identical steps outlined above.

To avoid unnecessary details, the results of the analysis to systematically cover all faulty 2-sequences using the mutants of the 1-Reg in Figure 9.2 are summarized. It appears that following REs are of particular interest when dealing with system vulnerabilities:

$$\begin{array}{c} (rgv^+)^*r,\ (rgv^+)^*rc,\ (rgv^+)^*rct^+,\ (rgv^+)^*rct^*o, \\ (rgv^+)^*rct^*og,\ (rgv^+)^*rct^*ogv^+,\ (rgv)^*rct^*ogv^+r \end{array} \tag{13}$$

These REs represent the event sequences which are possible prefixes that can be constructed by analyzing the expression (9). The test inputs can now be constructed as described in the last section, for example, *rgvrv* which can be generated as an instance of RE $(rgv^+)^*rv$ (11), which is extended from the sub RE in (11) of the RE in (9).

9.1.5 Conclusion

This section aims to extend, refine, and formalize event-based testing approaches based on the previous works [66][25][42][44][45] for modeling, testing and analysis of system vulnerabilities. The approach takes not only the desired, but also the undesired behaviors into account. Thus, a *holistic* view concerning the complete behavior of the system is preserved, which includes not only functional behavior but also a range of system vulnerabilities addressed by the attributes like safety, security, usability, etc. Incorporation of both the desired and undesired features of the system in the model allows a practical way to realize the "design for testability" in software design – a concept initially introduced in the seventies [168] for hardware. The degree of undesirability is represented in the form of a risk ordering relation – an expression of relative levels of risks posed by hazardous states. This allows targeting the design of tests at specific system attributes.

The key aspects of this work are (a) the extension of the previous approach by the use of mutant models which enable more precise and thorough handling of the faults, (b) the use of mutant selection strategies that increase the efficiency by reducing the number of generated mutants and test cases, and (c) the formalization of the concepts using a notation which is better suited for event-based testing.

The complete framework can be based on the concept of k-Regs. Since the approach relies on simple event-based modeling, it can be adapted in other software modeling approaches and tools such as statecharts [87][88] and UML [133][70]. This may require further research into modification or extension of the algorithms based on formal definitions of the new models. For example, one may need to consider particularly the problems related to state explosion, hierarchy, and concurrency [152][147], as well as the semantics of these new models. A major benefit of using models which have simple semantics is that they have a greater degree of analyzability.

However, keeping the model simple is also a limitation, because it limits the expressiveness. Therefore, the presented model can only make an approximation to the actual behavior of the system in some cases. Although, this can be considered as a simplification to reduce the testing costs in practice, the extension of the introduced concepts to more expressive models is a possible future research.

9.2 Model-Based Mutation Testing Using Different Types of Models

It is possible to adapt MBMT approach using different models. This section demonstrates two such exemplary adaptations using pushdown automata (PDA) and place/transition nets (PNs).

9.2.1 Model-Based Mutation Testing Using Pushdown Automata

Most of work on MBMT makes use of *regular models* such as FSMs, ESGs and RGs, which represent regular (type-3) languages in Chomsky hierarchy. However, such models can hardly represent the features whose behavior depends on previous states of the software, as exemplified in the following.

- Some software features save interim results and invoke other features. After completing the invoked features, they determine their subsequent behavior based on not only the results returned from the invoked features but also the interim results.
- Most software includes the feature to cancel recent operations and subsequently go back to the previous state, which is generally known as *undo* [159][36][37].

This section proposes an alternative MBMT framework using PDA that relate to context-free (type-2) languages. The main difference of a PDA from a 1-Reg (or an FSA) is that it contains a stack as the memory to keep the track of certain information related to the computation and, thus, PDA are more expressive.

9.2.1.1 Basic Notions

A *pushdown automata (PDA)* is a tuple $M = (S, E, G, T, S0, Z0, F)$ where

- S is a finite set of *states* (or *state alphabet*),
- E is a finite set of *events* (or *event alphabet*),
- G is a finite set of *stack symbols* (or *stack alphabet*),
- $T: S \times E \cup \{\varepsilon\} \times G \rightarrow U$ ($U \subseteq S \times G^*$ is finite) is the *transition function* (ε is the *empty string*),
- $S0 \in S$ is the *initial state*,
- $Z0 \in G$ is the *initial stack symbol*, and
- $F \subseteq S$ is the set of *final states*.

Transition function *T* receives as input a triple (*p, a, X*), where *p* is the current state, *a* is the event received in the current state, and *X* is the topmost stack symbol. The output of *T* is a finite set of pairs (*q, w*), where *q* is the new state and *w* is the string of stack symbols which replaces *X* at the top of the stack. Thus a *transition* can be represented by 5-tuple (*p, a, X, q, w*). A *read* operation occurs if $w = X$, a *pop* operation is performed if $w = \varepsilon$ and *Y* is *push*ed onto the stack if $w = YX$. Also, a PDA is *deterministic*, if it satisfies the following properties.

- $|T(p, a, X)| = 1$ for each $p \in S$, $a \in E \cup \{\varepsilon\}$ and $X \in G$.
- For each $p \in S$ and $X \in G$, if $T(p, \varepsilon, X) \neq \varnothing$ then $T(p, a, X) = \varnothing$ for every $a \in E$.

In this section, the PDA models used satisfy the following properties. (1) They are deterministic (with no ε-transitions). (2) $G-\{Z0\} \subseteq S$. (3) Every state is reachable from *S0* and a final state is reachable from each state.

Figure 9.5 shows an example PDA where $S = \{1, 2, 3, 4\}$, $E = \{a, b, c, d\}$, there are ten transitions, $G = \{0, 2, 3\}$, $S0 = 1$, $Z0 = 0$ and $F = \{4\}$. For example, when the PDA receives event *b* in State 3, it performs transition labeled by *2/3:2, 2*, that is, *(3, b, 2, 2, 3:2)*.

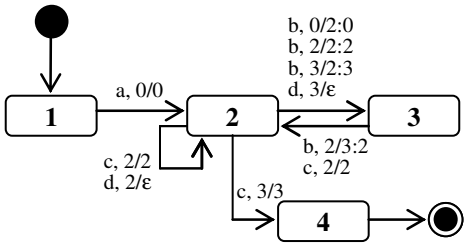


Figure 9.5. An example PDA.

A PDA has a stronger expressive power when compared to many other formal models. For example, a model which represents a regular (type-3) language in Chomsky hierarchy, results in an infinite state space while modeling a behavior represented by a PDA, unless the stack size is restricted (Since the states need to be defined as elements in $S \times G^*$ in order to represent the equivalent behavior). Even if the stack size is strictly restricted, the model may become too large to work with. Of course, models like UML state machine diagrams and UML profiles with action languages also have stronger representational powers [133]. However, they include informal representations or specific issues of programming languages, and thus need further formalizations or abstractions, respectively. More importantly, in their informal representation, they do not enable to use results of automata theory, which are very useful for test generation.

MBT aims to use simple models and increase the efficiency of the test process. PDA models are indeed simple in structure. Nevertheless, they can be applied to complex, real-life systems. This is feasible, because some of the real-life features are simplified; that is, measures are taken to abstract irrelevant to focus on relevant. This is common in MBT and explains the success of the widely accepted MBT techniques using simple, easy-to-understand approaches to be applied to practical systems. Here, more intrinsic features are captured using stronger, context-free PDA models, making the approach even more powerful than and, in certain cases, preferable to the use of regular models of simpler structure which are already applied to actual software system.

9.2.1.2 Mutation Operators

Similar to the discussion in Chapter 6, *marking*, *insertion* and *omission* mutation operators are defined for PDA. The combined and repeated applications of these operators can be used to transform a given PDA to any other PDA with the same event and stack alphabets.

Marking Operators. Given a PDA $M = (S, E, G, T, S0, Z0, F)$.

- *Mark state initial (Msti) operator* marks an existing state in M as the start state and the old start as a non-start.
- *Mark state final (Mstf) operator* marks an existing state in M as a final state.
- *Mark state nonfinal (Mstnf) operator* marks an existing final state in M as a non-final state.
- *Mark stack symbol initial (Msyi) operator* marks an existing stack symbol in M as the initial stack symbol. Old initial symbol is marked as non-initial.

Insertion Operators. Given a PDA $M = (S, E, G, T, S0, Z0, F)$.

- *Insert transition (Itr) operator* adds a new transition t to M. If $t = (q, a, X, p, w)$. It is assumed that $q,p \in S$, $a \in E \cup \{\varepsilon\}$, $X \in G$, $w \in G^*$ and $(p, w) \notin T(q, a, X)$. An insertion may also generate a non-deterministic PDA.
- *Insert state (Ist) operator* adds a new state q to M together with transitions $t_1, \ldots, t_k$. State q is not reachable from another state in M if no incoming non-loop transition to q is inserted. Furthermore, no state in M is reachable from state q if no outgoing non-loop transition from q is inserted.

Omission Operators. Given a PDA $M = (S, E, G, T, S0, Z0, F)$.

- *Omit transition (Otr) operator* deletes an existing transition t from M. If $t = (q, a, X, p, w)$, it is assumed that $q,p \in S$, $a \in E \cup \{\varepsilon\}$, $X \in G$, $w \in G^*$ and

(p, w) ∈ T(q, a, X). It is possible that an omission may leave some states with no incoming or outgoing transitions.

- *Omit state (Ost) operator* deletes an existing state *q* together with all the transition ingoing to and outgoing from *q*. After the deletion, some states in *M* may lose all their incoming or outgoing transitions.

Sometimes the mutation operators discussed above are too vague to use. To generate some specific faulty behavior, one needs only to modify or corrupt the existing transitions. In this way, the use of higher order mutations, which results in a huge number of mutants, can also be avoided. Thus, mutation operators that are relatively more precise and more suitable for PDA-based mutation testing or negative testing are defined as *replacement* mutation operators. The operators can also be seen as the controlled combinations of marking, insertion and omission mutation operators. They introduce specific faults into PDA models by corrupting the transitions without modifying the sets of states, events and stack symbols.

Write Replacement Operators. Given a PDA *M = (S, E, G, T, S0, Z0, F)*, *write replacement (Rw) operator* replaces the string to be put into the stack by the given string *w'*; that is, for *t = (p, a, X, q, w)*, *Rw(t, w') = (p, a, X, q, w')* where $w' \in G^* - \{w\}$. This operator can be performed in 4 different ways.

- *Replace with read (Rw-read) operator* replaces the stack operation associated to transition *t* with a read operation; that is, for *t = (p, a, X, q, w)*, *Rw-read(t) = (p, a, X, q, X)*. Note that the operator has no effect if *w = X*; that is, the operation is already a read operation. Therefore, this operator should only be performed on transitions where a non-read operation occurs.
- *Replace with push (Rw-push) operator* replaces the stack operation associated to transition *t* with a push operation. If the operation is already a push operation, a different string is pushed onto the stack. In other words, if *t = (p, a, X, q, wX)* and $w \in G^* - \{\varepsilon\}$, *Rw-push(t, w') = (p, a, X, q, w'X)* for some given $w' \in G^* - \{\varepsilon, w\}$. Otherwise, *Rw-push(t, w') = (p, a, X, q, w'X)* for some given $w' \in G^* - \{\varepsilon\}$.
- *Replace with pop (Rw-pop) operator* replaces the stack operation associated to transition *t* with a pop operation; that is, for *t = (p, a, X, q, w)*, *Rw-pop(t) = (p, a, X, q, ε)*. Note that this operator has no effect if *w = ε*; that is, the operation is already a pop operation.
- *Replace with pop-push (Rw-poppush) operator* replaces the stack operation associated to transition *t* with a pop followed by a push operation. More precisely, for *t = (p, a, X, q, w)*, *Rw-poppush(t, w') = (p, a, X, q, w')* for some given $w' \in G^*$, where $w' \notin \{\varepsilon, w''X\}$ for some $w'' \in G^*$. Note that if *w' = ε*, only a pop operation is performed, and if *w' = w''X* for some $w'' \in G^*$, either a read or a push operation is performed.

Read Replacement Operators. Given a PDA *M = (S, E, G, T, S0, Z0, F)*, *read replacement (Rr) operator* replaces the symbol on the top of the stack by the given symbol *X'*; that is, for *t = (p, a, X, q, w)*, *Rr(t, X') = (p, a, X', q, w)* where *X'∈ G-{X}*. This operator can also be performed in different ways.

- *Replace with initial stack symbol (Rr-init) operator* replaces the symbol read from stack in transition *t* with the initial stack symbol *Z0*; that is, for *t = (p, a, X, q, w)*, *Rr-init(t) = (p, a, Z0, q, w)*. Note that the operator has no effect if *X = Z0*; that is, top symbol is already initial stack symbol.
- *Replace with new stack top (Rr-top) operator* replaces the symbol read from stack in transition *t* with the new stack top; that is, for *t = (p, a, X, q, w),* where *w = Yw'*, *w' ∈ G** and *Y ∈ G*, *Rr-top(t) = (p, a, Y, q, w)*. This operator converts the operation in transition *t* to a push operation. Therefore, it is not applicable when a pop operation occurs; that is, *w = ε*, and has no effect if a push operation is performed; that is, *Y = X*.
- *Replace with another stack symbol (Rr-another) operator* replaces the symbol read from stack in transition *t* with a stack symbol other than initial stack symbol or the new stack top. More precisely, let *t = (p, a, X, q, w)*: If *w = ε*, *Rr-another(t, X') = (p, a, X', q, ε)* for some given *X' ∈ G-{Z0}*. Otherwise, *w = Yw'* for some *w' ∈ G** and *Y ∈ G*, *Rr-another(t, X') = (p, a, X', q, w'Y)* for some given *X' ∈ G-({Z0}∪{Y})*.

Event Replacement Operator. Given a PDA *M = (S, E, G, T, S0, Z0, F)*, *event replacement (Re) operator* replaces the event in a transition by another event; that is, for *t = (p, a, X, q, w)*, *Re(t, b) = (p, b, X, q, w)* where *b∈ E∪{ε}-{a}*.

Source Replacement Operator. Given a PDA *M = (S, E, G, T, S0, Z0, F)*, *source replacement (Rs) operator* replaces the source state in a transition by another state; that is, for *t = (p, a, X, q, w)*, *Rs(t, s) = (s, a, X, q, w)* where *s∈ S-{p}*.

Destination Replacement Operator. Given a PDA *M = (S, E, G, T, S0, Z0, F)*, *destination replacement (Rd) operator* replaces the destination state in a transition by another state; that is, for *t = (p, a, X, q, w)*, *Rd(t, s) = (p, a, X, s, w)* where *s∈ S-{q}*.

Figure 9.6 shows an example mutant of the PDA given in Figure 9.5, where transition *(3, c, 2, 4, 2)* is inserted.

9.2.1.3 Test Generation

N-switch transition coverage (fixed N≥0), which is developed based on the coverage for finite state machines [55], can be used to generate positive test cases from a given PDA. Its measuring object is a sequence of *N+1* successive

transitions containing stack top part (that is, some symbols on the top of the stack). More precisely:

A. The length of the transition sequences to be covered is *N+1* (or less if the transition sequences start from an initial state).

B. The length of the stack top part to be covered is *N*.

If the following condition is used in addition to A and B, the coverage criterion is called *N-switch faulty transition coverage (fixed N≥0).*

C. At least one faulty transition appears in each transition sequence.

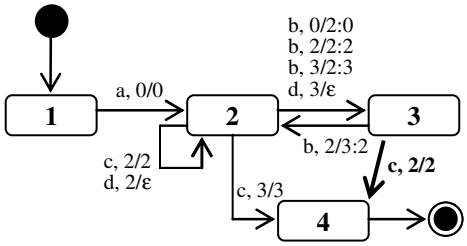


Figure 9.6. An example mutant of the PDA in Figure 9.5.

When N-switch faulty transition coverage criterion is satisfied, a test engineer can have the confidence that not only a suspicious operation is itself working correctly but also it is working correctly in the contexts induced by transition sequences where previous and following operations are also included. As the value of *N* gets larger, test engineers tend to have higher confidence in software quality, but then the size of the measuring objects (that is, the size of the test cases) also becomes larger.

Algorithm 9.1 outlines the steps to systematically generate test cases to satisfy N-switch faulty transition coverage for a given mutant PDA, which contains at least one faulty transition.

Algorithm 9.1 is developed based on depth-first search in directed graphs. It is obvious that a set of measuring objects is a subset of search objects. When a search object includes a faulty transition, it is identified as a measuring object. Covering all the search objects is indispensable for finding all the measuring objects since some measuring objects do not become executable unless specific search objects are previously executed. Consequently, it runs in $O(|S| (|T|/|S|)^{(N+1)})$ worst-case time, where $|S|$ is the number of states, $|T|$ is the number of transitions and *N* is from N-switch coverage. Also, it has a space complexity of $O(|S| (|T|/|S|)^{N})$.

Note that one can obtain positive test cases that satisfy N-switch transition coverage when Algorithm 9.1 is applied without Step 6 to an original PDA model.

Algorithm 9.1. PDA-Based Negative Test Generation

Step 1. Set the initial state as the current state, and begin to search the PDA.
Step 2. Select an executable outgoing transition in the current state.
If the execution of the selected transition results in the execution of a new search object, it is executed and is added to the test case under construction. Here a search object is a sequence of successive transitions with the stack top part that satisfies A and B given in Section 9.2.1.3.
If the selected transition does not result in a new search object, select another transition.
Step 3. Repeat step 2 until no new search object can be found.
Step 4. If there is a transition that is not selected in Step 2 (that is, a transition that has a possibility of deriving a new search object), backtrack to a previous state that has such a transition.
Step 5. Repeat from Step 2 to Step 5 similarly.
Step 6. If there is a test case that includes no measuring object, or there is a test case in which all the measuring objects are included in another test case, eliminate such a test case in order to derive a final set of test cases.

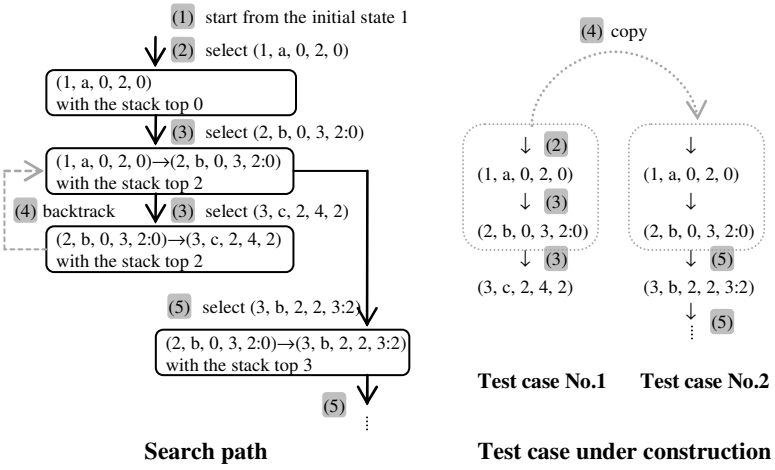


Figure 9.7. Example test case generation from the mutant PDA in Figure 9.6.

An example of test case generation from Figure 9.6 is shown in Figure 9.7. The nodes of the search path are search objects. Also, (1), (2), (3), (4) and (5) correspond to Steps 1, 2, 3, 4 and 5 of Algorithm 9.1, respectively.

Table 9.2. Overview of Test Generation from the PDA in Figure 9.5 and Figure 9.6

N	Number of Search/Measuring Objects		Number of Test Cases		Average Length of Test Cases	
	Positive	Negative	Positive	Negative	Positive	Negative
0	10 / 10	7 / 1	3	1	7.3	5.0
1	29 / 29	11 / 3	10	3	11.0	5.0
2	99 / 99	18 / 4	37	4	25.8	6.0
3	412 /421	29 / 8	171	8	85.7	7.3
4	1708 / 1708	44 / 10	714	10	364.3	8.2

Table 9.2 shows the overview of positive and negative test case generation from Figure 9.5 and Figure 9.6, respectively. As *N* becomes larger, the number of test cases increases significantly. For positive test cases, this increase is in general exponential. Therefore, in testing practice, *N*-switch based coverage criteria are used by selecting relatively small values for *N* to keep the overall testing process efficient and scalable.

9.2.2 Model-Based Mutation Testing Using Place/Transition Nets

PNs are also alternative models which can be used for MBMT. A PN is a kind of Petri net [140]; it is suitable to formally represent the behavior of concurrent software. PNs are already used for generation of positive test cases; they tend to yield too many test cases due to the large state space [160].

Alternatively, it is possible to adapt MBMT approach by defining fault states and generate negative test cases to cover these states. In this way, test generation over large state space can be avoided and discriminating test cases can be generated focusing on only specific fault states rather than covering the whole state space. To make this approach useful, a meaningful selection of mutants is quite important Thus, in addition to coverage criteria for adequacy of test cases, metrics for evaluating the mutant characteristics are also considered.

The approach introduced in this section differs from the one introduced by Fabbri et al. [72] in which the mutation operators are based on the fault model and fault classes introduced by Chow [55] for finite state machines. The approach discussed here makes use of the mutation operators that are based on two basic operators: *insertion* and *omission* as introduced by Belli et al. [40]. Furthermore, mutants are used to generate test cases for testing of the SUC, not for adequacy evaluation of given test sets.

9.2.2.1 Basic Notions

The structural elements of a PN are *places*, *tokens*, *arcs* and *transitions*. A *place* corresponds to a state of a component. A *token* corresponds to some kind of resource necessary for execution. An *arc* and a *transition* correspond to execution of a function. A state is represented as a *marking*, which is an array of the numbers of tokens each place contains. Figure 9.8 shows a simple example of a PN, which is a model of the producer-consumer problem. The PN contains six places *(p_1, p_2, …, p_6)* and four transitions *(t_1, t_2, t_3, t_4)*. p_2, p_4 and p_6 contain a token individually, and therefore the initial marking is described as [0,1,0,1,0,1].

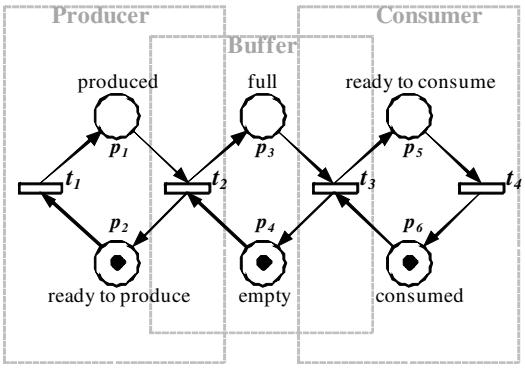


Figure 9.8. A PN of the producer-consumer problem.

A *fault state* in a PN is a principal fault and corresponds to a state that should not be reached in actual use. All fault states do not need to be defined comprehensively; only the principal ones should be defined. In Figure 9.8, fault states are determined as [0,1,1,1,0,1], [0,1,1,1,1,0], [1,0,1,1,0,1] and [1,0,1,1,1,0], since an incorrect state of the buffer can cause over-production or over-consumption.

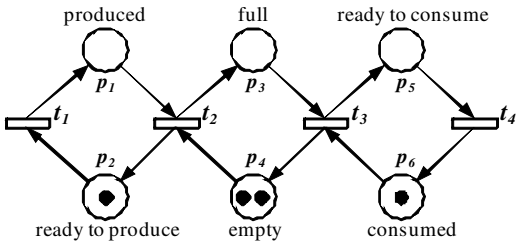


Figure 9.9. An effective mutant of the PN in Figure 9.8.

Mutation operators are applied to PN models to inject faults and to create mutants. However, not all mutants are useful during testing. Therefore, one needs to use some additional methods to select *effective mutants* that are, for example, not equivalent to another mutant PN or to the original PN and include at least one fault state. An effective mutant of Figure 9.8 is given in Figure 9.9. Also, Figure 9.10 shows a non-effective mutant. Evaluation of effectiveness is performed using *reachability graphs* generated from mutant PNs.

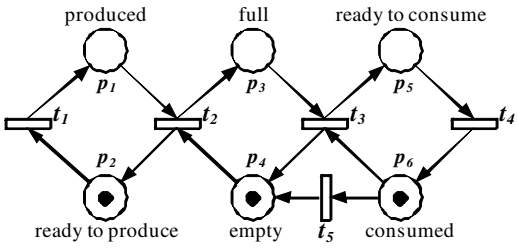


Figure 9.10. A noneffective mutant of the PN in Figure 9.8.

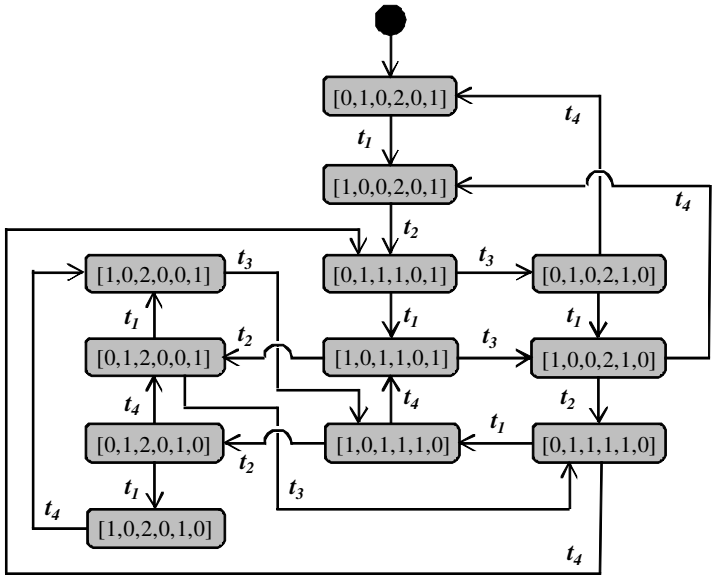


Figure 9.11. Reachability graph of the mutant in Figure 9.9.

A finite state machine which contains all the reachable markings in a PN is called a *reachability graph*. If the number of tokens increases infinitely, it is expressed by a symbol in the related markings in order to make the state space finite. As shown in Figure 9.11 and Figure 9.12, the mutant PNs in Figure 9.9 and Figure 9.10 are transformed into reachability graphs, respectively. The mutant in Figure 9.11 is called effective since its reachability graph includes all the fault states determined above. The effectiveness of mutants is evaluated based on the metrics introduced in Section 9.2.2.3.

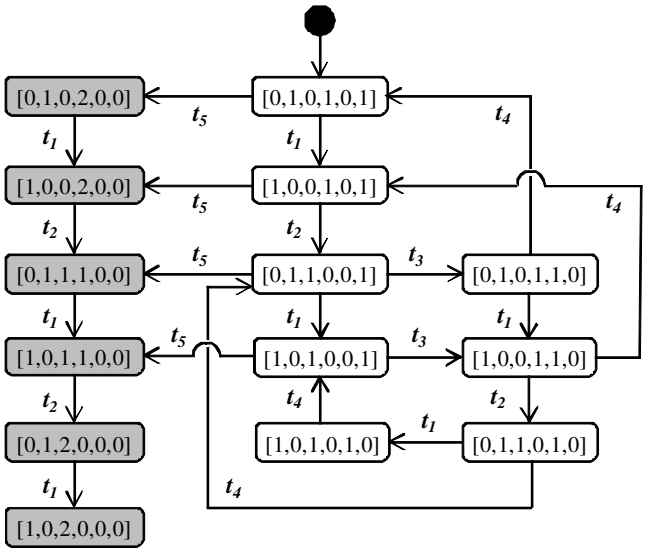


Figure 9.12. Reachability graph of the mutant in Figure 9.10.

A negative test case is called *effective* if it exercises a faulty behavior. The effectiveness of negative test cases is evaluated based on the metrics introduced in Section 9.2.2.4.

Based on the notions defined above, MBMT approach introduced in this study consists of the following five steps.

1. Construct a PN that represents the correct behavior of the SUT based on system specification.
2. Define fault states of SUT as markings in the PN.
3. Generate mutant PNs.
4. Generate the reachability graphs.
5. Select effective mutants and generate negative test cases.

9.2.2.2 Mutation Operators

Insertion and *omission* mutation operators are used for generation of PN mutants. They are are briefly described as follows.

Since a PN consists of tokens, arcs, transitions and places, one can define insertion operators as follows.

- *Insert token (Ito) operator* adds a token onto a place. An example is shown in Figure 9.8.
- *Insert arc (Ia) operator* adds an arc.
- *Insert transition (Itr) operator* adds a transition and arcs that connect this transition to existing places. Thus, insert arc operator is also used during a transition insertion. An example is given in Figure 9.9.
- *Insert place (Ip) operator* adds a place and arcs that connect this place to existing transitions. Thus, arc insertions are also performed during a place insertion.

Similarly, in a PN, one can omit tokens, arcs, transitions or places. Thus, omission operators are defined as follows.

- *Omit token (Oto) operator* deletes a token from a place
- *Omit arc (Oa) operator* deletes an arc.
- *Omit transition (Otr) operator* deletes a transition and arcs connected to this transition. Thus, arc omission operations are performed.
- *Omit place (Op) operator* deletes a place and arcs connected to this place. If the place initially has some tokens, they are also are omitted. Thus, a place omission includes the use of omit arc and omit operators.

9.2.2.3 Mutant Evaluation Metrics

Various mutants can arbitrarily be generated by using the mutation operators in Section 9.2.2.2. However, not all the generated mutants are effective. Thus, in order to generate effective negative test cases in MBMT, one needs to distinguish effective mutants from the ineffective ones. In the following, some metrics are introduced to evaluate the characteristics of mutants to ease the selection of effective mutants.

Fault state inclusion ratio (FSIR) indicates the percentage of fault states included in a mutant. With increasing FSIR, the mutant becomes more effective, since its ability to generate negative test cases increases. For example, the FSIR of the mutant whose reachability graph is shown in Figure 9.11 is 100% (4/4), since it contains all the fault states defined in Section 9.2.2.1, whereas, the FSIR of the mutant whose reachability graph is shown in Figure 9.12 is 0% (0/4), since it contains no fault states.

Unexpected state insertion ratio (USIR) indicates the ratio of unexpected states (that is, states that are not included in the original PN) to the total number of states in the original PN. With increasing USIR, the generated mutant contains more fault states including the ones that have possibly not been considered. Thus, mutants with such high ratio are preferred when various fault states are to be verified. Also, mutants with low ratio are preferred when the focus is only on the only the fault states that are explicitly defined. In Figure 9.8 and Figure 9.9, unexpected states are described as gray boxes. There are eight states in the original PN, described as white boxes in Figure 9.8. Thus, the USIR of the mutants in Figure 9.8 and Figure 9.9 are 150% (12/8) and 75% (6/8), respectively.

Expected state omission ratio (ESOR) indicates the ratio of expected states (that is, states in the original PN) that are omitted in a mutant to the total number of states in the original PN. With increasing ESOR, the mutant excludes more states from test generation. Thus, mutants with such low ratio are preferred when test engineers want to verify the relations between expected states and fault states. In addition, mutants with such high ratio are preferred when test engineers want to concentrate on faults that have great impacts. For example, the ESOR of the mutants in Figure 9.8 and Figure 9.9 are 100% (8/8) and 0% (0/8) respectively.

9.2.2.4 Coverage Criteria

Test coverage criteria or metrics to systematically generate effective negative test cases from the mutants are discussed below.

Fault state coverage ratio (FSCR) indicates the ratio of fault states included in the mutant and executed by negative test cases. With increasing FSCR, more fault states can be verified by executing the negative test cases. For example, assume that a negative test case $t_1 \rightarrow t_2 \rightarrow t_1 \rightarrow t_2$ that executes fault states [0,1,1,1,0,1] and [1,0,1,1,0,1] is generated from the graph in Figure 9.8. The graph contains four fault states. Therefore, the FSCR of the negative test case is 50% (2/4).

N-switch fault transition coverage ratio (N≥0) indicates the ratio of N+1 fault transition sequences (that is, sequences of successive transitions of length N+1 executing one or more fault states) in the mutant which are executed by negative test cases to all the maximum number of N+1 fault transition sequences. Note that N-switch fault transition coverage criterion subsumes fault state coverage. For example, in case of $N=0$, a negative test case $t_1 \rightarrow t_2 \rightarrow t_1 \rightarrow t_2$ generated from the graph in Figure 9.8 executes three 1-fault transition sequences [1,0,0,2,0,1]$\rightarrow t_2 \rightarrow$[0,1,1,1,0,1], [0,1,1,1,0,1]$\rightarrow t_1 \rightarrow$[1,0,1,1,0,1] and [1,0,1,1,0,1]$\rightarrow t_2 \rightarrow$[0,1,2,0,0,1]. The graph contains twelve 1-fault transition sequences. Therefore, the N-switch fault transition coverage ratio of the negative test case is 25% (3/12).

9.3 Extended Event-Based Models

In model-based testing, most of the recognized research work is based on state-based models [133][55][78][115][96][129]. However, the key features of systematic testing (predictability, controllability and observability) [2][151] are fulfilled by events. Thus, such models operate on "outputs" that are externally perceptible events and considered as semantic augmentation of the arcs that connect states. "States" cannot be observed and controlled directly from outside of the SUT, as they are controlled indirectly via event sequences. Therefore, state-based models function not due to their state orientation, but thanks to their proximity to event orientation.

One can argue to use state-based models and perform model-to-model transformations to obtain an event-based model. However, in practice, one tends to make different simplifications based on the selected representation and limitations. Various factors, like individual preferences, time/budget constraints and system characteristics, also affect these simplifications. Thus, an event-based model obtained using model-to-model transformations is expected to be different from another one built completely using event-centric approach in practice. Therefore, in many cases, the different focuses of the representations are rather more important than their equivalence.

This section discusses two types of extensions to the basic event-based model defined in Section 4: (1) One-sorted extensions and (2) many-sorted extensions. For the sake of simplicity, the discussion is carried out using visual representations. Furthermore, examples are outlined in order to demonstrate the use of the discussed extensions and their characteristics, and a list of aspects and ideas are given to speculate for further re-casting event-based modeling.

9.3.1 Extensions

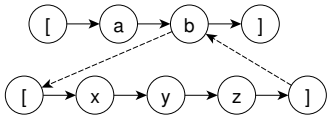
One-sorted extensions use a single, uniform syntax for the events; whereas. *many-sorted* extensions include events of different types using additional syntax for representing different meanings.

9.3.1.1 One-Sorted Extensions

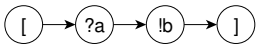
Over the years, needs for various systematic facilities have arisen to increase the expressive power of the models used in system/software engineering. The new facilities have sometimes come as extensions to the existing models (e.g., similar to derivation of timed automata from finite state automata), but in general, completely new representations have been introduced to meet these spontaneous needs (e.g., process algebra, or UML).

Thus, it is possible to extend k-Regs in different ways by considering the aforementioned formal representations introduced over the years. In the following, the basic k-Reg notion is extended following the traits below.

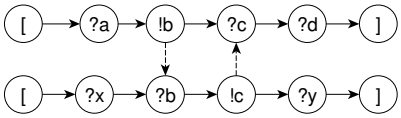
- Structure: Structured k-Regs
- Input-Output Labeling: Input-Output k-Regs
- Communication: Communicating k-Regs
- Quiescence: Quiescent k-Regs
- Time: Timed k-Regs
- Stack Component (Memory): Pushdown k-Regs



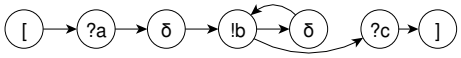
(a) A structured 1-Reg.



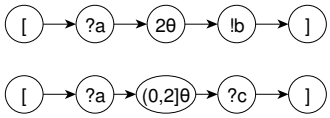
(b) An input-output 1-Reg.



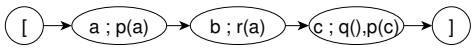
(c) A communicating 1-Reg.



(d) A quiescent 1-Reg.



(e) A timed 1-Reg



(f) A pushdown 1-Reg.

Figure 9.13. One-sorted k-Reg extensions.

Note that one can combine these traits to derive and use models such as "structured timed input-output k-Regs".

Structured k-Regs. *Structured k-Regs* enable further refinement of the events so that an event can represent a composite behavior that requires presence of multiple events; that is, the event is a composite event. For example, in Figure 9.13a, event *b* is a composite event representing another (sub) 1-Reg.

Naturally, it is possible to make other types of structuring as long as the elements of the structured node are compatible with the notion of event. An example of such structuring is the integration of decision tables [43].

Input-Output k-Regs. The basic k-Regs do not differ between different types of events; that is, user inputs and system outputs are represented by the same kind of nodes. In case the domain needs a differentiation between inputs and outputs, one can augment the semantics by additional symbols, for example, by "!" for inputs and "?" for outputs, leading to *input-output k-Regs*.

Communicating k-Regs. *Communicating k-Regs* combine k-Regs with sender-receiver structure. This is important for representation of synchronization of parallel behavioral k-Reg models. In Figure 9.13c, the dashed arcs represent the communications.

Quiescent k-Regs. A *quiescent k-Reg* includes the event δ for representation of no actions. Quiescence is important for continuation of the user actions after determining that there is no output from the system. An example is given in Figure 9.13d.

Timed k-Regs. *Timed k-Regs* are used to define an event-based model with respect to time; that is, a timed behavior is defined so that execution of an event is also dependent on time. Time is quantified using θ as the tick for time period, and intervals for time limits.

Pushdown k-Regs. In *pushdown k-Regs*, the model comes with a stack component. A sequence of stack operations is performed when an event is executed. The execution of the event is successful if and only if the related sequence of stack operations is also successful. For the sake of simplicity, only three stack operations are defined.

- *push – p(x)*: Writes symbol *x* to the top of the stack. This operation is always successful.
- *pop – q()* or *q(x)*: Reads and deletes the peek element *x* from the stack. *q()* fails if the stack is empty, and q*(x)* fails if the stack is empty or the peek element is not *x*.

- *peek – r()* or *r(x)*: Reads the peek element from the stack. *r()* fails if the stack is empty, and r*(x)* fails if the stack is empty or the peek element is not *x*.

9.3.1.2 Many-Sorted Extensions

Many sorted extensions employ some traits which are quite different from the ones used for one-sorted extensions. Generally, these traits are very application-specific.

The best examples of many-sorted extensions are event flow graphs (EFGs), where the events have different syntax and semantics for modeling of graphical user interfaces (GUIs). For example, in the EFG in Figure 9.14, the diamond-node is a menu-open event, the double-circle-node is a restricted-focus event, rectangle-nodes are termination events and circle-nodes are system-interaction events.

For more sophisticated event-centric modeling, several variations of EFGs have been created in recent work: (1) *Event Interaction Graphs (EIGs)* [124] that represent a subset of events in the system, and hence, are more compact and scalable, (2) *Event Semantic Interaction Graphs (ESIGs)* [174] that model a subset of "follows" relations – between events that are shown to interact at a semantic level, and (3) *Probabilistic EFGs (PEFGs)* [51] that form Bayesian networks and n-gram Markov models.

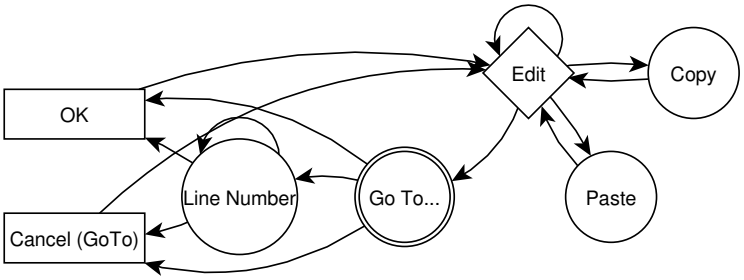


Figure 9.14. A many-sorted k-Reg extension (an EFG).

9.3.2 Examples

In this section, a simplified online conference initiation example is used to relay the ideas and to demonstrate how the aforementioned extensions can be used in practice.

In the online conference initiation, there is a user entity which would like to participate in a specific conference. Each user logs in the system, makes a join request and waits for the acceptance. After the acceptance is received, the user becomes a participant in the conference. Of course, a request can also be declined during initiation. In this case the user is allowed make another request. There is also an administrator entity which accepts or declines the participation requests and determines which users are allowed to take part in the conference.

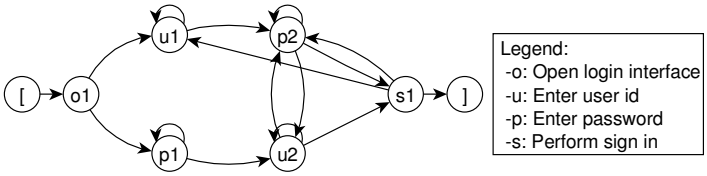


Figure 9.15. A basic 1-Reg for `Login` interface.

During modeling, the abstraction level is quite important and allows the tester to create different models to test for different purposes. For example, one can create and use the (basic) 1-Reg in Figure 9.15 to model the events and their sequences in user login scenario.

During login process, a user enters user id and password pair in any order. The sign in is activated, if only both user id and password are entered. Furthermore, after sign in is executed, it can either succeed or fail. In case of failure, if password is wrong then user only needs to enter a new password. However, if user id is wrong both user id and password need to be entered again.

In Figure 9.15, event *o* opens the login interface, where events *u* and *p* are for entering the user id and password, respectively. Also, event *s* corresponds to sign in event. Note that multiple instances of *u* and *p* events are used as contexted events to leave out infeasible event sequences. For example, *u1* cannot be followed by *s1*, where as *u2* can be.

9.3.2.1 One-Sorted Extension Examples

This section gives examples for some of the one-sorted extensions introduced in Section 9.3.1.1.

9.3.2.1.1 A Structured 1-Reg using a Decision Table

Note that the 1-Reg in Figure 9.15 takes the order of entering *u* and *p* events into account while modeling. However, if there are too many of such events in a

system, the number of orderings grows very fast. Therefore, one may choose to ignore the order that the data is entered. In such cases, decision tables can be used to structure and simplify the model. Figure 9.16 demonstrates such a model for the 1-Reg in Figure 9.15.

In Figure 9.16, *up* represents the event for entering user id and password. The decision table suggests that event *s* follows *up* if and only if both *u* and *p* are performed.

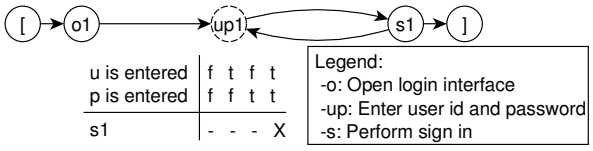


Figure 9.16. A structured 1-Reg for `Login` interface.

9.3.2.1.2 An Input-Ouput 1-Reg

It is clear that the 1-Reg in Figure 9.15 contains no information on system outputs; the events are contexted based on the set of following input events. Therefore, it can only be used to generate test cases and in combination with non-output based test oracles, such as sequence-based or language-based test oracles, without any additional information.

When outputs are also included, the number of existing events may grow and further contexted events may emerge. Figure 9.17 demonstrates the input-output 1-Reg for the login interface.

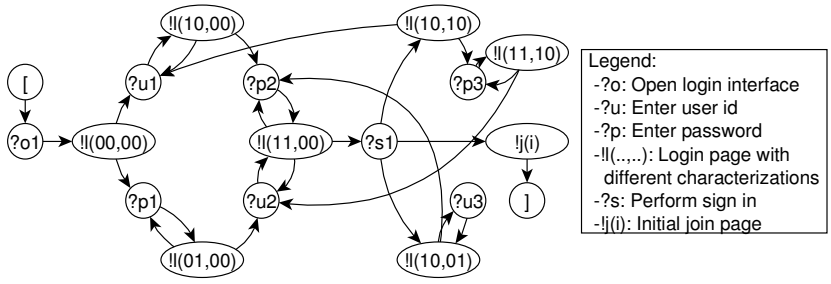


Figure 9.17. An input-output 1-Reg for `Login` interface.

In Figure 9.17, "?" are used to label input events and "!" are used to label output events. Most of the output events are of the form $!l(U_eP_e,U_wP_w)$. Here, l signifies login interface, U signifies user id and P signifies password. Also, U_eP_e signifies whether user id and password are entered or not. It can take the values of 00, 01, 10 and 11 (for example, U_eP_e=00 means that both user id and password are empty, whereas U_eP_e=11 shows that they are entered). Furthermore, U_wP_w can only take the values of 00, 10 and 01, and shows if there is a warning message on user id or password, or not (for example, if U_wP_w=10, there is a warning message on user id, and if U_wP_w=01, an incorrect password is used). There is also a single output called $!j(i)$ which signifies that the login is successful and initial join interface is displayed.

Figure 9.17 demonstrates that explicit inclusion of outputs tends to increase the complexity of the model (however provides a more precise and complete picture). Therefore, one can use the 1-Reg in Figure 9.15 to generate test cases and the 1-Reg in Figure 9.17 to derive the expected outputs for these sequences. Of course, one can also choose not include the outputs explicitly in the model but instead associate (or embed) them to (or into) each event in the model. In this case, the (basic) 1-Reg in Figure 9.18 can be used.

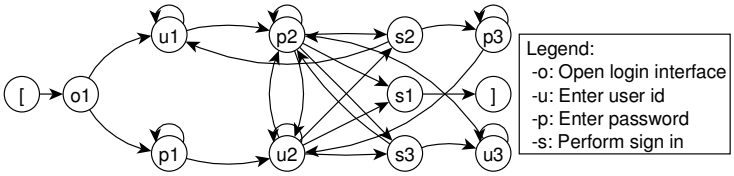


Figure 9.18. Another basic 1-Reg for `Login` interface.

Note that, since outputs are also considered in addition to event-sequences during contexting, the 1-Reg in Figure 9.18 is quite different from the 1-REG in Figure 9.15. For example, s is indexed 3 times (*s1*, *s2* and *s3*), because each of them has a different output and can be followed by different set of events. For similar reasons, events *u3* and *p3* are also included in this model.

9.3.2.1.3 A Structured 1-Reg using a Sub Model

Using one of the login 1-Regs presented above as a sub model, a conference initiation 1-Reg can be constructed. Figure 9.19 demonstrates the corresponding structured input-output 1-Reg.

In Figure 9.19, l refers to the IO 1-Reg for login interface (Figure 9.17). Therefore, it is a composite event. Furthermore,?r is the event for making the

request for participating in a conference. There are two possible outputs: !*j(d)* for a declined and !*j(a)* for an accepted request. Upon acceptance, the initiation of a user ends.

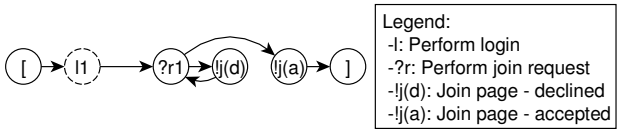


Figure 9.19. A structured input-output 1-Reg for user initiation.

As demonstrated by the 1-Reg in Figure 9.19, not only basic 1-Regs but also input-output 1-Regs (and other types of 1-Regs) can be structured.

9.3.2.1.4 A Communicating 1-Reg

Note that the initiation of a user does not solely depend on what a user does; it also depends on the response of the conference administrator. Thus, one may need to consider the user and the administrator together. For this purpose, communicating input-output 1-Regs can be used. Figure 9.20 demonstrates such a 1-Reg.

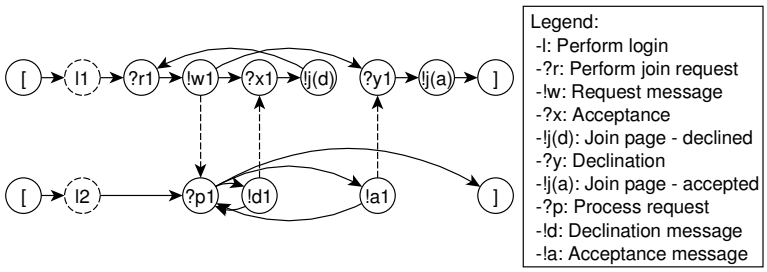


Figure 9.20. A communicating input-output 1-Reg for user initiation.

In Figure 9.20, the top 1-Reg is for the user side and the bottom 1-Reg is for the administrator side. An administrator logs in like a user by performing event *l2* (Figure 9.17). However, after logging in, the administrator processes the conference participation requests (?*p*), and accepts (!*a*) or declines (!*d*) them. Therefore, inputs to the user initiation are not only controlled by the user but also by the administrator. Furthermore, some outputs of user initiation are observed by

the administrator. For this reason, some input and output events, which are internal from a user's perspective, are made explicit.

In Figure 9.20, the output event which is not directly observable by the user is !*w*. It represents a request message sent by user to the administrator. Furthermore, the input events which are controlled by the administrator are ?*x* and ?*y*. These events are activated after receiving, respectively, a declination or an acceptance from the administrator.

9.3.2.1.5 A Quiescent 1-Reg

Note that Figure 9.20 also shows us that in absence of a functioning administrator, there is no response. Therefore, events ?*x* or ?*y* cannot commence. In such cases, to specify the lack of actions or outputs, one can use quiescent 1-Regs. Figure 9.21 shows such a 1-Reg. For simplicity, it outlines the lack of response from user perspective without including the administrator. Therefore, it is derived from the 1-Reg in Figure 9.19 (not Figure 9.20).

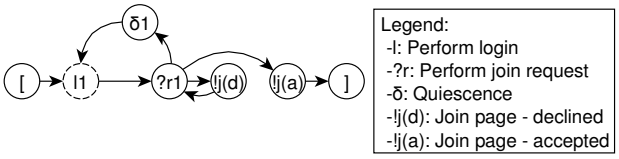


Figure 9.21. A quiescent input-output 1-Reg for user initiation.

In Figure 9.21, event δ signifies that "*r* is a quiescent event; that is, after a join request is performed, there might be a lack of response (input or output events). In this case, user is required to log in and try to make a join request once again.

9.3.2.1.6 A Timed 1-Reg

In practice quiescence can be realized or handled using time-outs. For example, after performing a join request, one can wait for a specified time and then either perform another join request or continue with the next step if there is a response from the administrator.

In Figure 9.22, after performing a join request, a time-out event is executed. After 2θ time, if there is no response from the administrator, ?*r* is performed again. Otherwise, depending on the type of response (declination or acceptance), either !*j(d)* or !*j(a)* follows.

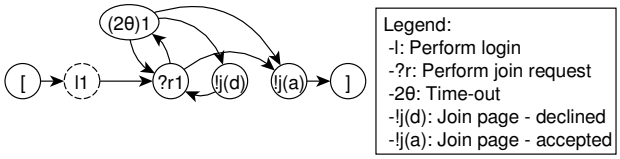


Figure 9.22. A timed input-output 1-Reg for user initiation.

9.3.2.1.7 A Pushdown 1-Reg

Assume that the login interface whose basic 1-Reg model is given in Figure 9.15 contains go-back-events using which the user is allowed to take back previously performed events, and so go-back. The 1-Reg extensions mentioned so far are not strong enough to fully capture such behaviors, because an additional component, that is, a stack, is required to keep the track of previous events. Figure 9.23 demonstrates a pushdown 1-Reg model where together with each event a sequence of operations is performed on the stack in order to keep or restore previous events.

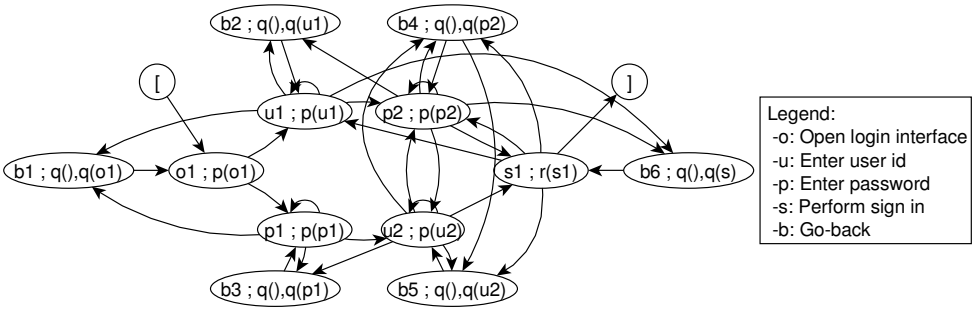


Figure 9.23. A pushdown 1-Reg for `Login` interface.

In Figure 9.23, there are additional *b* events which are called as go-back events. Each non go-back event except performs a push operation to keep track of events, and each go-back event performs two subsequent pop operations to cancel the previously performed event. For example, assume that event *u2* is executed after event *p1*. In this case, the peek element in the stack is *u1* and the second one is *u2*. Thus, one can only execute *b3* to return back to the non go-back event which comes before the last non go-back event *u2*. *b4* and *b5* cannot be executed because their related stack operations fail, and *b1*, *b2* and *b6* cannot be executed because there are no edges from *u2* to these go-back events.

9.3.2.1.8 A Combined Extension 1-Reg

Finally, in order to demonstrate how the set of traits discussed in Section 9.3.1.1 can be combined (and extended) to build a new model that meets the needs of a specific type of application, an administrator model is constructed assuming that it is a web-service (composition). Figure 9.24 demonstrates the corresponding 1-Reg model for the administrator; it is similar to ESG4WSC [41].

In Figure 9.24, an event can be either public or private, depending on whether it can be performed or observed by the user directly. Therefore, "?" and ":" are used to label public and private input events, and "!" and "." are used to label public and private outputs events, respectively. Here, first, the user makes a request to the administrator web service (?*r*). Later, if the user id format is valid, the service verifies the user (:*v*), e.g., it can call another web service to do this. If the verification is successful (.*s*), the service returns an acceptance response (!*a*). If it fails (.*f*), the service returns declination response (!*d*). Also, the service returns invalid user id response (!*i*) upon a request with invalid user id.

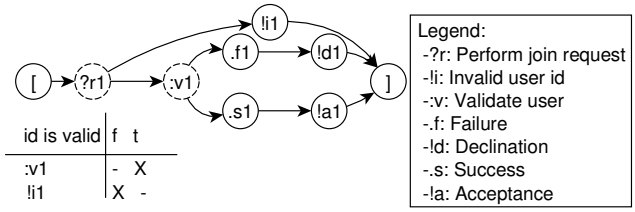


Figure 9.24. A combined extension 1-Reg for administrator web service.

9.3.2.2 Many-Sorted Extension Examples

This section demonstrates examples for some of the many-sorted extensions introduced in Section 9.3.1.2.

9.3.2.2.1 An Event Flow Graph

Assume that the login interface is a GUI where *o* opens the interface and *s* closes it. Also, while entering a user id or a password the underlying system makes checks in order to enable or disable event *s*, and performing *s* closes the login interface interacting with the system. In this case, the model in Figure 9.15 gets some syntactical changes depending on the semantics of each event residing in it, and Figure 9.25 is constructed.

In Figure 9.25, *o* is a menu open event, and thus represented using a diamond-shaped node. The remaining events are all system interaction events and they are

represented using circle nodes. Note that *s* is given in a circle node, although it is also a termination event.

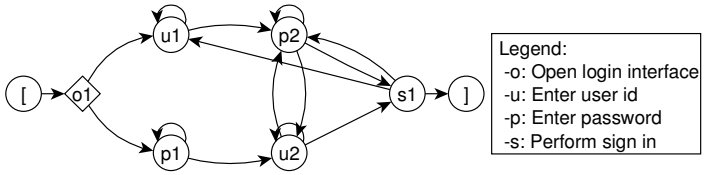


Figure 9.25. An EFG for `Login` interface.

9.3.2.2.2 An Event Interaction Graph

In GUI testing, one often chooses to focus on system interaction and termination events (assuming that other events are not fault-prone), and interactions between them. For this purpose, EIGs can be used. Figure 9.26 shows the EIG of the EFG in Figure 9.25.

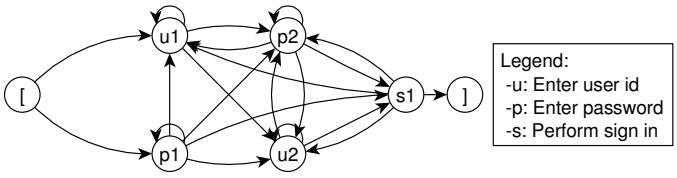


Figure 9.26. An EIG for `Login` interface.

In Figure 9.26, arcs do not form a *follows* relation anymore. They simply show that an event is *reachable* from one another. For example, arc *(p1, s1)* does not exist in the EFG (Figure 9.25). However, since there is a path from *p1* to *s*1, it is included in the EIG (Figure 9.26).

9.3.3 Conclusion

Based on sound results of research and experience, this section aims to promote event-centric models, and takes a step to enable their use in testing of various different types of systems and establish a classification. To do this, the model introduced in Chapter 4 is employed as the basis model allowing the use of mathematical, sound methods introduced in Chapters 4, 5, 6 and 7.

Note that the traditional flow graph model of computer programs can also be considered as an event model, where the events are the executions of the statements or linear blocks of statements. Therefore, the traditional adequacy criteria (including control flow coverage and data flow coverage criteria) can be easily adapted for more general context of software testing. In addition, further research could be directed to more complicated situations of event-driven systems, such as distributed testing architectures and testing concurrent and non-deterministic systems. In such situations, it may be possible to regard othey types of models as event-based models. Thus, adequacy criteria can be defined for such models and techniques for generation of test cases can be adapted for them.

10 Conclusion and Outlook

MBT operates on models by abstracting from irrelevant details of the system. This makes MBT an attractive approach because it helps to increase fault detection and costs effectiveness and efficiency. MBMT introduces the use fault models or mutants in test generation for the purpose of detecting certain faults modeled by the mutants.

The MBMT approach proposed in this work is event-based. It uses a new event-based grammar model which is equivalent to RGs, FSA and REs. As opposed to the existing MBT or MBMT approaches which use fixed models, the proposed approach systematically transforms a given model to generate morphologically different models. This enables the generation of test cases covering longer event sequences so that one can exercise different test paths to check the correct functioning of the test object with respect to the expected behavior. Also, with the use of event-based mutation operators, the set of faults under consideration needs not be finite anymore; more precisely, by varying the model morphology, one can extend the set of fault models and, thus, incrementally select finite subsets of this possibly infinite set of faults in a systematic way. Since mutants are used for test generation, the approach makes use of certain mutant selection strategies to cut back the immense number of mutants that can be generated, without disregarding any relevant faults.

The major novelties of the approach can be outlined as follows.

- Equivalent mutants and multiple mutants modeling the same faults are excluded without even being generated; the existing approaches either do not care about these mutants or exclude only the equivalent mutants after generating and comparing them to the original model.
- Any mutant can be killed by a unique, dedicated test case that is generated in linear time without comparing the mutant against the original model; the existing approaches either use the whole mutant for coverage-based test generation or compare the mutant against the original models to generate a discriminating test case.

- The associated set of fault models can systematically be extended to consider different, subtle faults; the existing approaches use fixed models and the number of mutants considered is finite.

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≥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. Thus, the project budgetary costs can be adjusted by varying *k*. To the authors' knowledge, no other approach combines these advantages.

The characteristics of the approach are analyzed, and comparisons against random testing and ESG-based testing approaches are 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.

- When compared to ESG-based and MK-based MBMT in terms of the number of generated mutants, k-Reg-based testing approach (for *k=1,2,3*) generates and use only 0.40% to 3.62% of the mutants generated by the ESG-based and the MK-based approaches.
- On the average, when compared to random testing where same test targets are covered,
  - *k-Reg* detected 2% to 14% fewer faults than random testing while saving 13% to 67% of the effort (that is, it detected 13% to 147% more faults per executed event); *M-k-Reg* detected 0% to 5% more faults than random testing while saving 13% to 23% of the effort (that is, it detected 16% to 29% more faults per executed event).
  - for *k-Reg*, the increase in the number of revealed faults is up to ~1.06 times higher and, for *M-k-Reg*, up to 28%.
  - for both *k-Reg* and *M-k-Reg*, the increase in the test generation times is around 99% less.
  - for *k-Reg*, the increase in the number of executed events is up to 67% less and, for *M-k-Reg*, up to 17%.
- On the average, when compared to ESG-based MBMT by balancing the test execution effort,
  - *k-Reg* detected 34% to 74% more faults than ESG-based MBMT (and it detected 34% to 79% more faults per executed event); *M-k-Reg*

detected 37% to 74% more faults than ESG-based MBMT (and it detected 37% to 73% more faults per executed event).
  - for *k-Reg*, the increase in the number of revealed faults is up to 91% higher and, for *M-k-Reg*, up to 76%.
  - for *k-Reg* and *M-k-Reg*, the increase in the test generation times is up to 99.9% less.
- On the average, when compared to MK-based by balancing the test execution effort,
  - *k-Reg* detected 72% to 105% more faults than MK-based MBMT (and it detected 73% to 134% more faults per executed event); *M-k-Reg* detected 76% to 106% more faults than MK-based MBMT (and it detected 76% to 115% more faults per executed event).
  - for *k-Reg*, the increase in the number of revealed faults is up to 3.2 times higher and, for *M-k-Reg*, up to 1.2 times.
  - for *k-Reg* and *M-k-Reg*, the increase in the test generation times is up to 99.96% less.

Further perspectives are also considered by

- applying the approach for modeling, testing and analysis of system vulnerabilities,
- proposing novel MBMT approaches based on pushdown automata and place/transition nets, which are not purely event-oriented, and
- extending the event-based models to prepare a basis for further improvements to the event-based MBMT approaches.

Further work on the subject can be carried out to extend the morphology variation, mutant generation and test generation aspects to these extended event-based models. This can also be performed for the testing approaches which use models that are not purely event-oriented, because such models are still used in practice. Furthermore, state-based or other features of these models may come in handy, especially when a certain level of internal information about the system can be integrated into the model and used in testing.

Bibliography

[1] H. Aboelfotoh, O. Abou-Rabia, O., H. Ural, "A test generation algorithm for systems modelled as non-deterministic FSMs," Software Engineering Journal, vol.8, no.4, Jul 1993, pp. 184-188.

[2] M. Abramovici, M.A. Breuer, A.D. Friedman, Digital Systems Testing and Testable Design, Computer Science Press, 1990.

[3] A.T. Acree, On Mutation, Ph.D. Thesis, Georgia Institute of Technology, 1980.

[4] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, F.G. Sayward, "Mutation Analysis," Technical Report, DTIC Document, Sep. 1979.

[5] K. Adamopoulos, M. Harman, R. Hierons, "How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution," AAAI Genetic and Evolutionary Computation Conference 2004 (GECCO 2004), Lecture Notes in Computer Science, vol. 3103, Springer-Verlag, 26-30 Jun. 2004, pp. 1338-1349.

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

[7] A. Aho, A. Dahbura, D. Lee, M. Uyar, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours," IEEE Transactions on Communications, vol. 39, no. 11, Nov. 1991, pp. 1604-1615.

[8] B.K. Aichernig, "Mutation Testing in the Refinement Calculus," Formal Aspects of Computing Journal, vol. 15, no. 2-3, Nov. 2003, pp. 280-295.

[9] B.K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, "Model-Based Mutation Testing of Hybrid Systems," 8th International Conference on Formal Methods for Components and Objects (FMCO 2009), Lecture Notes in Computer Science, vol. 6286, F.S. de Boer, M.M. Bonsangue, S. Hallerstede, M. Leuschel, Eds., Springer-Verlag, 2010, pp. 228-249.

[10] B.K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, "Efficient Mutation Killers in Action," Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST 2011), IEEE Computer Society, 21-25 Mar. 2011, pp. 120-129.

[11] B.K. Aichernig, J. He, "Mutation testing in UTP," Formal Aspects of Computing, vol. 21, no. 1-2, pp. 33–64, 2009.

[12] B.K. Aichernig, E. Jöbstl, "Towards Symbolic Model-Based Mutation Testing: Combining Reachability and Refinement Checking," Proceedings of the 7th Workshop on Model-Based Testing (MBT 2012), arXiv, 2012, pp. 88-102.

[13] B.K. Aichernig, E. Jöbstl, "Towards Symbolic Model-Based Mutation Testing: Pitfalls in Expressing Semantics as Constraints," Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST 2012), IEEE Computer Society, 17-21 Apr. 2012, pp. 752-757.

[14] B.K. Aichernig, E. Jöbstl, "Efficient Refinement Checking for Model-Based Mutation Testing," Proceedings of the 12th International Conference on Quality Software (QSIC 2012), IEEE Computer Society, 27-29 Aug. 2012, pp. 21-30.

[15] B.K. Aichernig, E. Jöbstl, M. Kegele, "Incremental Refinement Checking for Test Case Generation," Test and Tools, Lecture Notes in Computer Science, vol. 7942, Springer, 2013, pp. 1-19.

[16] R. Alur, C. Courcoubetis, M. Yannakakis, "Distinguishing tests for nondeterministic and probabilistic machines," Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC 1995), ACM, 1995, pp. 363-372.

[17] P.E. Ammann, P.E. Black, W. Majurski, "Using Model Checking to Generate Tests from Specifications," Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM 1998), IEEE Computer Society, 1998, pp. 46-54.

[18] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008.

[19] J.H. Andrews, L.C. Briand, Y. Labiche, "Is mutation an appropriate tool for testing experiments?," Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), IEEE Computer Society, 15-21 May 2005, pp. 402-411.

[20] A. Arcuri, L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), IEEE Computer Society, 21-28 May 2011, pp. 1-10.

[21] A. Arcuri, L. Briand, "Formal Analysis of the Probability of Interaction Fault Detection Using Random Testing," IEEE Transactions on Software Engineering, vol. 38, no. 5, Sept.-Oct. 2012, pp. 1088-1099.

[22] A. Arcuri, M.Z. Iqbal, L. Briand, "Random Testing: Theoretical Results and Practical Implications," IEEE Transactions on Software Engineering, vol. 38, no. 2, Mar.-Apr. 2012, pp. 258-277.

[23] M. von der Beeck, "A Comparison of Statecharts Variants," Formal Techniques of Real-Time and Fault-Tolerant Systems (FTRTFT 1994), Lecture Notes in Computer Science, vol. 863, Springer, 1994, pp. 128-148.

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

[25] F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces," Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001), IEEE Computer Society, 27-30 Nov. 2001, pp. 34-43.

[26] F. Belli, M. Beyazıt, "A Formal Framework for Mutation Testing," Proceedings of the 2010 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010), IEEE Computer Society, 9-11 Jun. 2010, pp. 121-130.

(Corrected version is available at http://adt.et.upb.de/download/papers/BB2010_SSIRI2010corrected.pdf.)

[27] F. Belli, M. Beyazıt, A. Hollmann, M. Linschulte, S. Padberg, "Ereignis-basierter Test grafischer Benutzeroberflächen – ein Erfahrungsbericht," Softwaretechnik Trends, Gesellschaft für Informatik (GI), vol. 30, no. 2, Nov. 2010, pp. 21-23. (In German)

[28] F. Belli, M. Beyazıt, "Event-Based Mutation Testing vs. State-Based Mutation Testing - An Experimental Comparison," Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference (COMPSAC 2011), IEEE Computer Society, 18-22 Jul. 2011, pp. 650-655.

[29] F. Belli, M. Beyazıt, "Grammar-Based Mutation Testing," Proceedings of the 5th Turkish National Software Engineering Symposium (V. Ulusal Yazılım Mühendisliği Sempozyumu - UYMS 2011), 26-28 Sep. 2011, pp. 38-45. (in Turkish)

[30] F. Belli, M. Beyazıt, "Using Regular Grammars for Event-Based Testing," Proceedings of the 18th International Conference on Implementation and Application of Automata (CIAA 2013), Lecture Notes in Computer Science, vol. 7982, S. Konstantinidis, Ed., Springer, Heidelberg, 16-19 Jul. 2013, pp. 48-59.

[31] F. Belli, M. Beyazıt, "Mutant Selection for Event-Based Testing," Proceedings of the 7th Turkish National Software Engineering Symposium (VII. Ulusal Yazılım Mühendisliği Sempozyumu - UYMS 2013), 25-28 Sep. 2013, CEUR-WS, http://ceur-ws.org/Vol-1072. (in Turkish)

[32] F. Belli, M. Beyazıt, N. Güler, "Event-Based GUI Testing and Reliability Assessment Techniques - An Experimental Insight and Preliminary Results," Proceedings of the 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2011 / TESTBEDS 2011), IEEE Computer Society, 21-25 Mar. 2011, pp. 212-221.

[33] F. Belli, M. Beyazıt, N. Güler, "Event-Oriented, Model-Based GUI Testing and Reliability Assessment—Approach and Case Study," Advances in Computers, vol. 85, A. Memon, Ed., 2012, pp. 277-326.

[34] F. Belli, M. Beyazıt, A.P. Mathur, N. Nissanke, "Modeling, Analysis and Testing of System Vulnerabilities," Advances in Computers, vol. 84, A. Hurson, S. Sedigh, Eds., 2012, pp. 39-92.

[35] F. Belli, M. Beyazıt, A. Memon, "Testing is an Event-Centric Activity," Proceedings of the 6th International Conference on Software Security and Reliability (SERE-C 2012), IEEE Computer Society, 20-22 Jun. 2012, pp. 198-206.

[36] F. Belli, M. Beyazıt, T. Takagi, Z. Furukawa, "Mutation Testing of "Go-Back" Functions Based on Pushdown Automata," Proceedings of the 2011 IEEE 4th International Conference on Software Testing, Verification and Validation (ICST 2011), IEEE Computer Society, 21-15 Mar. 2011, pp. 249-258.

[37] F. Belli, M. Beyazıt, T. Takagi, Z. Furukawa, "Model-based Mutation Testing Using Pushdown Automata," IEICE Transactions on Information and Systems, vol. E95-D, no. 9, 9 Sep. 2012, pp. 2211-2218.

[38] F. Belli, C.J. Budnik, "Minimal spanning set for coverage testing of interactive systems," First International Colloquium on Theoretical Aspects of Computing (ICTAC 2004), Lecture Notes in Computer Science, vol. 3407, Springer-Verlag, Sep. 2004, pp. 220-234.

[39] F. Belli, C.J. Budnik, L. White, “Event-based modeling, analysis and testing of user interactions: Approach and Case Study,” Journal of Software Testing, Verification and Reliability (STVR), vol. 16, no. 1, Mar. 2006, pp. 3-32.

[40] F. Belli, C.J. Budnik, W.E. Wong, “Basic operations for generating behavioral mutants,” Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION 2006), IEEE Computer Society, 7-10 Nov. 2006, pp. 9-18.

[41] F. Belli, A.T. Endo, M. Linschulte, A. Simao, “Model-based testing of web service compositions,” IEEE 6th International Symposium on Service Oriented System Engineering (SOSE2011), IEEE Computer Society, 12-14 Dec. 2011, pp.181-192.

[42] 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.

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

[44] F. Belli, N. Nissanke, C.J. Budnik, “Finite-State Modeling, Analysis and Testing of System Vulnerabilities - Approach and Case Study,” Technical Report, Angewandte Datentechnik TR 2003/5, University of Paderborn, 2003.

[45] F. Belli, N. Nissanke, C.J. Budnik, A.P. Mathur, “Test Generation Using Event Sequence Graphs,” Technical Report, Angewandte Datentechnik TR 2005/6, University of Paderborn, Aug. 2005.

[46] M. Beyazıt, T.S. Deistler, N. Gökçe, “Event-Based Mutation Testing vs. State-Based Mutation Testing – Comparison Using a Web-based System,” Modellbasiertes Testen und Testautomatisierung (MOTES 2010), GI Jahrestagung (2), Lecture Notes in Informatics, Gesellschaft für Informatik (GI), 2010, pp. 327-332.

[47] P.E. Black, V. Okun, Y. Yesha, “Mutation operators for specifications,” Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000), IEEE Computer Society , 2000, pp. 81-88.

[48] P.E. Black, V. Okun, Y. Yesha, “Mutation of model checker specifications for test generation and evaluation,” Mutation Testing for the New Century, Kluwer International Series on Advances in Database Systems, vol. 24., W.E. Wong, Ed., Kluwer Academic Publishers, 2001, pp. 14-20.

[49] S. Boroday, A. Petrenko, R. Groz, “Can a Model Checker Generate Tests for Non-Deterministic Systems?,” Electronic Notes in Theoretical Computer Science, vol. 190, no. 2, 31 Aug. 2007, pp. 3-19.

[50] L. Briand, Y. Labiche, “A UML-Based Approach to System Testing,” Software and System Modeling, vol. 1, no. 1, 1 Sep. 2002, pp. 10-42.

[51] P. Brooks, A.M. Memon, “Automated GUI Testing Guided by Usage Profiles,” Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), ACM, 2007, pp. 333-342.

[52] T.A. Budd, A.S. Gopal, “Program Testing by Specification Mutation,” Computer Languages, vol. 10, no. 1, 1985, pp. 63-73.

[53] T.Y. Chen, Y.T. Yu, "On the expected number of failures detected by subdomain testing and random testing," IEEE Transactions on Software Engineering, vol. 22, no. 2, Feb. 1996, pp. 109-119.

[54] N. Chomsky, "Three models for the description of language," IRE Transactions on Information Theory, vol.2, no.3, Sep. 1956, pp. 113-124.

[55] 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.

[56] V. Cortellessa, A. Di Marco, P. Inverardi, Model-Based Software Performance Analysis, Springer, 2011.

[57] J. Davies, Specification and Proof in Real-Time CSP, Cambridge University Press, 1993.

[58] M.E. Delamaro, J.C. Maldonado, A.P. Mathur, "Interface Mutation: An Approach for Integration Testing," IEEE Transactions on Software Engineering, vol. 27, no. 3, Mar. 2001, pp. 228-247.

[59] 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.

[60] R. Dorofeeva, K. El-Fakih, S. Maag, A.R. Cavalli, N. Yevtushenko, "Experimental evaluation of FSM-based testing methods," Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), IEEE Computer Society, 7-9 Sep. 2005, pp. 23-32.

[61] R. Dorofeeva, K. El-Fakih, N. Yevtushenko, "An Improved Conformance Testing Method," Formal Techniques for Networked and Distributed Systems (FORTE 2005), Lecture Notes in Computer Science, vol. 3731, F. Wang, Ed., Springer-Verlag, 2005, pp. 204-218.

[62] R.R. Dumke, R. Braungarten, M. Blazey, H. Hegewald, D. Reitz, K. Richter, "Structuring software process metrics," Proceedings of the 16th International Workshop on Software Metrics and DASMA Software Metrik Kongress (IWSM/MetriKon 2006), Shaker Verlag GmbH, 2006, pp. 483-497.

[63] R.R Dumke, R. Braungarten, M. Kunz, A. Schmietendorf, C. Wille, "Strategies and appropriateness of software measurement frameworks," Proceedings of the International Conference on Software Process and Product Measurement (MENSURA 2006), 2006, pp. 150-170.

[64] J.W. Duran, S.C. Ntafos, "An Evaluation of Random Testing," IEEE Transactions on Software Engineering, vol. SE-10, no.4, Jul. 1984, pp. 438-444.

[65] J. Edmonds, E.L. Johnson, "Matching, Euler Tours and the Chinese Postman," Mathematical Programming, vol. 5, no. 1, 1 Dec. 1973, pp. 88-124.

[66] B. Eggers, F. Belli, "A Theory on Analysis and Construction of Fault-Tolerant Systems," Informatik-Fachberichte 84, Springer, 1984, pp. 139-149. (in German)

[67] H. Ehrig, H. Kreowski, U. Montanari, G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation, Volumes 1–3, World Scientific Publishers, 1996.

[68] A.T. Endo, A. Simão, "Experimental Comparison of Test Case Generation Methods for Finite State Machines," Proceedings of the 5th International Conference on Software Testing, Verification and Validation Workshops (ICST 2012 / A-MOST 2012), IEEE Computer Society, 17-21 Apr. 2012, pp. 549-558.

[69] A.T. Endo, A. Simão, "Evaluating Test Suite Characteristics, Cost, and Effectiveness of FSM-based Testing Methods," Information and Software Technology, Available online 21 Jan. 2013.

[70] H.E. Eriksson, M. Penker, B. Lyons, D. Fado, UML 2 Toolkit, Wiley, 2004.

[71] S.C.P.F. Fabbri, J.C. Maldonado, M.E. Delamaro, P.C. Masiero, "Mutation Analysis Testing for Finite-State Machines," Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE 1994), IEEE Computer Society, 6-9 Nov. 1994, pp. 220-229.

[72] S.C.P.F. Fabbri, J.C. Maldonado, P.C. Masiero, M.E. Delamaro, E. Wong, "Mutation Testing Applied to Validate Specifications based on Petri Nets," Proceedings of the IFIP TC6 8th International Conference on Formal Description Techniques, G. von Bochmann, R. Dssouli, O. Rafiq, Eds., Chapman & Hall, Ltd., 1995, pp. 329-337.

[73] S.C.P.F. Fabbri, J.C. Maldonado, T. Sugeta, P.C. Masiero, "Mutation Testing Applied to Validate Specifications Based on Statecharts," Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE 1999), IEEE Computer Society, 1999, pp. 210-219.

[74] A. Farooq, An Evaluation Framework for Software Test Processes, Ph.D. Thesis, University of Magdeburg, 2009.

[75] A. Farooq, R.R. Dumke, A. Schmietendorf, H. Hegewald, "A classification scheme for test process metrics," Proceedings of South East European Software Testing Conference (SEETEST 2008), dpunkt.verlag, 2008.

[76] M. Felleisen, "On the expressive power of programming languages," Science of Computer Programming, vol. 17, no. 1–3, Dec. 1991, pp. 35-75.

[77] P.G. Frankl, E.J. Weyuker, "A formal analysis of the fault-detecting ability of testing methods," IEEE Transactions on Software Engineering, vol. 19, no. 3, Mar. 1993, pp. 202-213.

[78] S. Fujiwara; G.v. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, "Test selection based on finite state models," IEEE Transactions on Software Engineering, vol. 17, no. 6, Jun. 1991, pp. 591-603.

[79] G. Fraser, F. Wotawa, "Nondeterministic Testing with Linear Model-Checker Counterexamples," Proceedings of the 7th International Conference on Quality Software (QSIC 2007), IEEE Computer Society, 11-12 Oct. 2007, pp. 107-116.

[80] V.K. Garg, M.T. Ragunath, "Concurrent regular expressions and their relationship to Petri nets," Theoretical Computer Science, vol. 96, no. 2, 13 Apr. 1992, pp. 285-304.

[81] A.S. Gopal, T.A. Budd, "Program Testing by Specification Mutation," Technical Report, TR 83-17, University of Arizona, 1983.

[82] S. Gossens, "Enhancing System Validation with Behavioral Types," Proceedings of the 7th International Symposium on High Assurance Systems Engineering (HASE 2002), IEEE Computer Society, 2002, pp. 201-208

[83] G. Gönenç, "A Method for the Design of Fault Detection Experiments," IEEE Transactions on Computers, vol. C-19, no. 6, Jun. 1970, pp. 551-558.

[84] B.J.M. Grün, D. Schuler, A. Zeller, "The Impact of Equivalent Mutants," International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2009), IEEE Computer Society, 1-4 Apr. 2009, pp. 192-199.

[85] D. Hamlet, "Foundation of Software Testing: Dependability Theory," Proceedings of the 2nd Symposium on Foundations of Software Engineering (SIGSOFT 1994), ACM, Dec. 1994, pp. 128-139.

[86] R.G. Hamlet, "Testing Programs with the Aid of a Compiler," IEEE Transactions on Software Engineering, vol. SE-3, no. 4, Jul. 1977, pp. 279-290.

[87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," Science of Computer Programming, vol. 8, no. 3, Jun. 1987, pp. 231-274.

[88] D. Harel, A. Namaad, "The STATEMATE Semantics of Statecharts," ACM Transactions on Software Engineering and Methodology, vol. 5, no. 4, Oct. 1996, pp. 293-333.

[89] M. Harman, Y. Jia, W.B. Langdon, "Strong higher order mutation-based test data generation," In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of Software Engineering (ESEC/FSE 2011), ACM, 2011, pp. 212-222.

[90] M.J. Harrold, A.J. Offutt, K. Tewary, "An approach to fault modeling and fault seeding using the program dependence graph," Journal of Systems and Software, vol. 36, no. 3, Mar. 1997, pp. 273-295.

[91] F.C. Hennine, "Fault detecting experiments for sequential circuits," Proceedings of the 5th Annual Symposium on Switching Circuit Theory and Logical Design, IEEE Computer Society, 11-13 Nov. 1964, pp. 95-110.

[92] R.M. Hierons, "Adaptive Testing of a Deterministic Implementation Against a Nondeterministic Finite State Machine," The Computer Journal, vol, 41, no. 5, 1998, pp. 349-355.

[93] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A.J.H. Simons, S. Vilkomir, M.R. Woodward, H. Zedan, "Using formal specifications to support testing," ACM Computing Surveys, vol. 41, no. 2, Article 9, Feb. 2009, 76 pages.

[94] R.M. Hierons, M. Harman, S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," Software Testing, Verification and Reliability, vol. 9, no. 4, 1999, pp. 233-262.

[95] R.M. Hierons, H. Ural, "Optimizing the length of checking sequences,"IEEE Transactions on Computers, vol. 55, no. 5, May 2006, pp. 618- 629.

[96] R.M. Hierons, H. Ural, "Generating a checking sequence with a minimum number of reset transitions," Automated Software Engineering, vol. 17, no. 3, 2010, pp. 217-250.

[97] Y. Hirshfeld, "Petri nets and the equivalence problem," Computer Science Logic (CSL 1993), 7th Workshop, Selected Papers, Lecture Notes in Computer Science, vol. 832, E. Börger, Y. Gurevich, K. Meinke, Eds., Springer, 13-17 Sep. 1993, pp. 165-174.

[98] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[99] A. Hollmann, Model-Based Mutation Testing for Test Generation and Adequacy Analysis, Ph.D. Thesis, University of Paderborn, 2011.

[100] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages and Computation, 3rd Edition, Addison-Wesley, 2006.

[101] I. Hwang, T. Kim, S. Hong, J. Lee, "Test selection for a nondeterministic FSM," Computer Communications, vol. 24, no. 12, 15 Jul. 2001, pp. 1213-1223.

[102] J.I. Ianow, "Logic Schemes of Algorithms," Problems of Cybernetics I, 1958, pp. 87-144. (in Russian)

[103] Institute of Electrical and Electronics Engineers, "IEEE Standard for System and Software Verification and Validation - Redline," IEEE Std 1012-2012 - Redline, 25 May 2013, http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6251988.

[104] Y. Jia, M. Harman, "An Analysis and Survey of the Development of Mutation Testing," IEEE Transactions on Software Engineering, vol. 37, no. 5, Sep.-Oct. 2011, pp. 649-678.

[105] K. Jensen, N. Wirth, Pascal, User Manual and Report, P.B. Hansen, D. Gries, C. Moler, G. Seegmüller, N. Wirth, Eds., Springer-Verlag, 1974.

[106] D.K. Kaynar, N. Lynch, R. Segala, F. Vaandrager, "Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems," Proceedings of the 24th Real-Time Systems Symposium (RTSS 2003), IEEE Computer Society, 3-5 Dec. 2003, pp. 166-177.

[107] S.W. Kim, J.A. Clark, J.A. McDermid, "Investigating the Effectiveness of Object-Oriented Testing Strategies with the Mutation Method," Journal of Software Testing, Verification and Reliability, vol. 11, no. 4, Dec. 2001, pp. 207-225.

[108] Y.G. Kim, H.S. Hong, D.H. Bae, S.D. Cha, "Test Cases Generation from UML State Diagrams," IEE Proceedings - Software, vol. 146, no. 4, Aug. 1999, pp. 187-192.

[109] P. Klint, R. Lämmel, C. Verhoef, "Toward an engineering discipline for grammarware," ACM Transactions on Software Engineering and Methodology, vol. 14, no. 3, Jul. 2005, pp. 331-380.

[110] H. Kloosterman, "Test Derivation from Non-Deterministic Finite State Machines" Proceedings of the IFIP TC6/WG6.1 5th International Workshop on Protocol Test Systems (IWPTS 1992), North-Holland Publishing Co., 1992, pp. 297-308.

[111] B. Korel, "Automated Test Data Generation for Programs with Procedures," Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1996), ACM, 1996, pp. 209-215.

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

[113] M.E. Kramer, J. Klein, J.R.H. Steel, B. Morin, J. Kienzle, O. Barais, J.-M. Jézéquel, "On the Formalisation of GeKo: a Generic Aspect Models Weaver," Technical Report, UL-CHAPTER-2012-280, SNT, University of Luxembourg, 2012.

[114] M.K. Kwan, "Graphic Programming Using Odd or Even Points," Chinese Math., vol. 1, no. 3, 1962, pp. 273-277.

[115] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - a survey," Proceedings of the IEEE, vol. 84, no. 8, Aug. 1996, pp. 1090-1123.

[116] N.G. Leveson, "Software Safety: Why, What, and How," ACM Computing Surveys, vol. 18, no. 2, Jun. 1986, pp. 125-163.

[117] N.G. Leveson, Safeware, System Safety and Computers, Addison-Wesley, 1995.

[118] G. Luo, A. Petrenko, G. v. Bochmann, "Selecting test sequences for partially-specified nondeterministic finite state machines," Proceedings of the IFIP TC6/WG6.1 7th International Workshop on Protocol Test Systems (IWPTS 1994), T. Mizuno, T. Higashino, N. Shiratori, Eds., Chapman & Hall, Ltd., Nov. 1994, pp. 95-110.

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

[120] A. Masood, R. Bhatti, A. Ghafoor, A.P. Mathur, "Scalable and Effective Test Generation for Role-Based Access Control Systems," IEEE Transactions on Software Engineering, vol. 35, no. 5, Sep.-Oct. 2009, pp. 654-668.

[121] A. Masood, A. Ghafoor, A.P. Mathur, "Conformance Testing of Temporal Role-Based Access Control Systems," IEEE Transactions on Dependable and Secure Computing, vol. 7, no. 2, Apr.-Jun. 2010, pp. 144-158.

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

[123] G.H. Mealy, "A Method for Synthesizing Sequential Circuits," Bell Systems Technical Journal, vol. 34, Sep. 1955, pp. 1045-1079.

[124] A.M. Memon, Q. Xie, "Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software," IEEE Transactions on Software Engineering, vol. 31, no. 10, Oct. 2005, pp. 884-896.

[125] T. Mens, P. Van Gorp, "A Taxonomy of Model Transformation," Electronic Notes in Theoretical Computer Science, vol. 152, 27 Mar. 2006, pp. 125-142.

[126] R. Milner, Communications and Concurrency, Prentice Hall, 1989.

[127] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, 1996.

[128] E.F. Moore, "Gedanken Experiments on Sequential Machines," Automata Studies, Annals of Mathematical Studies, vol. 34, Princeton University Press, 1956, pp. 129-153.

[129] S. Mouchawrab, L.C. Briand, Y. Labiche, "Assessing, Comparing, and Combining Statechart-based testing and Structural testing: An Experiment," Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM2007), IEEE Computer Society, 2007, pp. 41-50.

[130] J. Myhill, "Finite Automata and the Representation of Events," Technical Report, WADD TR 57-624, Wright Patterson AFB, 1957, pp. 112-137.

[131] N. Nissanke, H. Dammag, "Design for Safety in Safecharts with Risk Ordering of States," Safety Science, vol. 40, no. 9, Dec. 2002, pp. 753-763.

[132] S.C. Ntafos, "A comparison of some structural testing strategies," IEEE Transactions on Software Engineering, vol. 14, no. 6, Jun. 1988, pp. 868-874.

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

[134] A.J. Offutt, P. Ammann, L. Liu, "Mutation Testing Implements Grammar-Based Testing," Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION 2006), IEEE Computer Society, 7-10 Nov. 2006, pp. 12-21.

[135] A.J. Offutt, J.H. Hayes, “A semantic model of program faults,” Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1996), S.J. Zeil, W. Tracz, Eds., ACM, 1996, pp. 195-200.

[136] A.J. Offutt, J. Pan, “Automatically detecting equivalent mutants and infeasible paths,” Software Testing, Verification and Reliability, vol. 7, no. 3, 1997, pp. 165-192.

[137] V. Okun, P.E. Black, Y. Yesha, “Testing with model checker: Insuring fault visibility,” Proceedings of WSEAS International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems, N.E. Mastorakis, P. Ekel, Eds., WSEAS, Oct. 2002, pp. 1351-1356.

[138] J. Paakki, “Attribute grammar paradigms—a high-level methodology in language implementation,” ACM Computing Surveys, vol. 27, no. 2, Jun. 1995, pp. 196-255.

[139] F. Pereira, D. Warren, “Definite clause grammars for language analysis,” Readings in Natural Language Processing, B.J. Grosz, K. Sparck-Jones, B.L. Webber, Eds., Morgan Kaufmann Publishers, 1985, pp. 101-124.

[140] J.L. Peterson, Petri Net Theory and the Modelling of Systems, Prentice-Hall, 1981.

[141] A. Petrenko, A. Simão, N. Yevtushenko, “Generating Checking Sequences for Nondeterministic Finite State Machines,” Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST 2012), IEEE Computer Society, 17-21 Apr. 2012, pp. 310-319.

[142] A. Petrenko, N. Yevtushenko, “Testing from partial deterministic FSM specifications,” IEEE Transactions on Computers, vol. 54, no. 9, Sept. 2005, pp. 1154- 1165.

[143] A. Petrenko, N. Yevtushenko, G.v. Bochmann, “Testing Deterministic Implementations from their Nondeterministic Specifications”, Proceedings of the IFIP TC6/WG6.1 9th International Workshop on Testing of Communicating Systems (IWTCS 1996), 1996, pp. 125-140.

[144] A. Petrenko, N. Yevtushenko, A. Lebedev, A. Das, “Nondeterministic State Machines in Protocol Conformance Testing,” Proceedings of the IFIP TC6/WG6.1 6th International Workshop on Protocol Test Systems (IWTCS 1993), O. Rafiq, Ed., North-Holland Publishing Co., 1993, pp. 363-378.

[145] R.E. Prather, “Regular Expressions for Program Computations,” The American Mathematical Monthly, vol. 104, no. 2, Feb. 1997, pp. 120-130.

[146] P. Purdom, “A Sentence Generator for Testing Parsers,” BIT Numerical Mathematics, vol. 12, no. 3, 1 Sep. 1972, pp. 366-375.

[147] S.C.V. Raju, A. Shaw, “A Prototyping Environment for Specifying, Executing and Checking Communicating Real-Time State Machines,” Software - Practice and Experience, vol. 24, no. 2, Feb. 1994, pp. 175-195.

[148] W. Reisig, Petri Nets: An Introduction, Springer-Verlag, 1985.

[149] J. Rumbaugh, I. Jacobson, G. Booch, Unified Modeling Language Reference Manual, 2nd Edition, Pearson Higher Education, 2004.

[150] A. Salomaa, I.N. Sneddon, Theory of Automata, Pergamon Press, 1969.

[151] K.I. Satish, “Tutorial on design for testability (DFT) "An ASIC design philosophy for testability from chips to systems",” Proceedings of the 6th Annual IEEE International ASIC Conference and Exhibit, 1993, pp. 130-139.

[152] F.B. Schneider, “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, ACM Computing Surveys,” vol. 22, no. 4, Dec. 1990, pp. 299-319.

[153] A. Simão, A. Petrenko, N. Yevtushenko, “Generating Reduced Tests for FSMs with Extra States,” In Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International Workshop on Formal Approaches to Testing of Software, Lecture Notes in Computer Science, vol. 5826, M. Núñez, P. Baker, M.G. Mercedes, Springer-Verlag, 2009, pp. 129-145.

[154] A. Simão, A. Petrenko, “Fault Coverage-Driven Incremental Test Generation,” The Computer Journal, vol. 53, no. 9, Nov. 2010, pp. 1508-1522.

[155] E.G. Sirer, B.N. Bershad, “Using Production Grammars in Software Testing,” Proceedings of the 2nd Conference on Domain-Specific Languages (DSL 1999), ACM, 3-5 Oct. 1999, pp. 1-13.

[156] P.A. Stocks, Applying Formal Methods to Software Testing, PhD Thesis, University of Queensland, 1994.

[157] N. Storey, Safety-critical Computer Systems, Addison-Wesley, 1996.

[158] T. Sugeta, J.C. Maldonado, W.E. Wong, “Mutation Testing Applied to Validate SDL Specifications,” Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems, Lecture Notes in Computer Science, vol. 2978, Springer-Verlag, Mar. 2004, pp. 193-208.

[159] T. Takagi, Z. Furukawa, “The Pushdown Automaton and Its Coverage Criterion for Testing Undo/Redo Functions of Software,” The 9th International Conference Computer and Information Science (ICIS 2010), IEEE Computer Society, 18-20 Aug. 2010, pp. 770-775.

[160] T. Takagi, N. Oyaizu, Z. Furukawa, “Concurrent N-switch Coverage Criterion for Generating Test Cases from Place/Transition Nets,” Proceedings of the 2010 IEEE/ACIS 9th International Conference on Computer and Information Science (ICIS 2010), IEEE Computer Society, 18-20 Aug. 2010, pp. 782-787.

[161] T. Takagi, R. Takata, Z. Furukawa, F. Belli, M. Beyazıt, “Metrics for Model-Based Mutation Testing Based on Place/Transition Nets,” Proceedings of the Joint Conference of the 21st International Workshop on Software Measurement (IWSM) and the 6th International Conference on Software Process and Product Measurement (Mensura) - Fast Abstracts, IEEE Computer Society, 3-4 Nov. 2011, pp. 7-10.

[162] R.D. Tennent, Specifying Software, Cambridge University Press, Cambridge, 2002.

[163] G.J. Tretmans, “Test Generation with Inputs, Outputs and Repetitive Quiescence,” Technical Report, TR-CTIT-96-26, Centre for Telematics and Information Technology University of Twente, Enschede, 1996.

[164] P. Tripathy, K. Naik, “Generation of Adaptive Test Cases from Nondeterministic Finite State Models,” Proceedings of the IFIP TC6/WG6.1 5th International Workshop on Protocol Test Systems (IWPTS 1992), North-Holland Publishing Co., 1992, pp. 309-320.

[165] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., 2006.

[166] M.P. Vasilevskii, "Failure diagnosis of automata," Cybernetics and Systems Analysis, vol. 9, no. 4, 1 Jul. 1973, pp. 653-665.

[167] M. Weiglhofer, B. Aichernig, F. Wotawa. "Fault-based Conformance Testing in Practice," International Journal of Software and Informatics, vol. 3, no. 2-3, Jun./Sep. 2009, pp. 375-411.

[168] T.W. Williams, K.P. Parker, "Design for Testability - A Survey," IEEE Transactions on Computers, vol. 31, no. 1, Jan. 1982, pp. 2-15.

[169] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, 2000.

[170] A. Wood, "Software Reliability Growth Models," Technical Report, 96-1, Tandem Computers Inc. - Corporate Information Center, Sep. 1996.

[171] Q. Xie, A.M. Memon, "Using a Pilot Study to Derive a GUI Model for Automated Testing," ACM Transactions on Software Engineering Methodology, vol. 18, no. 2, Nov. 2008, pp. 1-35.

[172] N. Yevtushenko, A. Petrenko, "Synthesis of Test Experiments in Some Classes of Automata," Automatic Control and Computer Sciences, vol. 24, no. 4, Apr. 1991, pp. 50-55.

[173] N. Yevtushenko, A. Lebedev, A. Petrenko, "On Checking Experiments with Nondeterministic Automata," Automatic Control and Computer Sciences,"vol. 24, no. 6, 1991, pp. 81-85.

[174] X. Yuan, A.M. Memon, "Using GUI Run-Time State as Feedback to Generate Test Cases," Proc. 29th International Conference on Software Engineering (ICSE 2007), IEEE Computer Society, May 2007, pp. 396-405.

[175] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, H. Mei, "Test generation via Dynamic Symbolic Execution for mutation testing," In Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM 2010), IEEE Computer Society, 12-18 Sep. 2010, pp. 1-10.

[176] L. Zheng, D. Wu, "A Sentence Generation Algorithm for Testing Grammars," Proceedings of the 33rd International Computer Software and Applications Conference (COMPSAC 2009), vol. 1, IEEE Computer Society, 20-24 Jul. 2009, pp. 130-135.

[177] Z.Q. Zhou, D.H. Huang, T.H. Tse, Z. Yang, H. Huang, T.Y. Chen, "Metamorphic Testing and Its Applications," Technical Report, TR-2004-12, The University of Hong Kong, 2004.

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

[179] H. Zhu, B. Yu, "An Experiment with Algebraic Specifications of Software Components," Proceedings of the 10th International Conference on Quality Software (QSIC 2010), IEEE Computer Society, 14-15 Jul. 2010, pp. 190-199.

List of Figures

List of Tables

List of Algorithms

Part IV

Appendices

A Models for Case Studies

A.1 Productions of ShearBar 1-Reg Model

1. S → eventInitialization c(eventInitialization)
2. c(eventAAdjustment) → eventTimeoutLeft_A1 c(eventTimeoutLeft_A1) | eventKnockingRight_A1 c(eventKnockingRight_A1)
3. c(eventTimeoutLeft_A1) → ε
4. c(eventTimeoutRight_A1) → ε
5. c(eventALeftEngineDeparted200_3) → eventARightEngineDeparted200AndRightDistanceLess c(eventARightEngineDeparted200AndRightDistanceLess) | eventARightEngineDe-parted200AndLeftDistanceLess c(eventARightEngineDeparted200AndLeftDistanceLess)
6. c(eventARightEngineDeparted200AndRightDistanceLess) → eventARightEngineApproached324_1 c(eventARightEngineApproached324_1)
7. c(eventARightEngineDeparted200AndLeftDistanceLess) → eventALeftEngineApproached324_1 c(eventALeftEngineApproached324_1)
8. c(eventALeftEngineApproached324_1) → eventALeftEngineApproached324_2 c(eventALeftEngineApproached324_2)
9. c(eventARightEngineApproached324_1) → eventARightEngineApproached324_2 c(eventARightEngineApproached324_2)
10. c(eventALeftEngineDeparted162_1) → eventARightEngineApproached324_3 c(eventARightEngineApproached324_3)
11. c(eventARightEngineDeparted162_1) → eventALeftEngineApproached324_3 c(eventALeftEngineApproached324_3)
12. c(eventALeftEngineDepartedRef01) → eventARightEngineDepartedRef01AndNoKnocking c(eventARightEngineDepartedRef01AndNoKnocking) | eventARightEngineDepartedRef01AndKnocking c(eventARightEngineDepartedRef01AndKnocking)
13. c(eventARightEngineDepartedRef01AndNoKnocking) → ε
14. c(eventKnockingLeft_A1) → eventTimeoutRight_A1 c(eventTimeoutRight_A1) | eventALeftEngineDe-parted200_1 c(eventALeftEngineDeparted200_1)
15. c(eventKnockingRight_A1) → eventTimeoutRight_A1 c(eventTimeoutRight_A1) | eventKnockingLeft_A1 c(eventKnockingLeft_A1)

16. c(eventARightEngineDepartedRef01AndKnocking) →
    eventARightEngineDeparted50AndKnocking_1
    c(eventARightEngineDeparted50AndKnocking_1) |
    eventARightEngineDeparted50AndNoKnocking
    c(eventARightEngineDeparted50AndNoKnocking)
17. c(eventALeftEngineDeparted50AndKnocking_1) →
    eventARightEngineDeparted50AndNoKnocking
    c(eventARightEngineDeparted50AndNoKnocking) |
    eventARightEngineDeparted50AndKnocking_2
    c(eventARightEngineDeparted50AndKnocking_2)
18. c(eventARightEngineDeparted50AndKnocking_1) →
    eventALeftEngineDeparted50AndKnocking_1
    c(eventALeftEngineDeparted50AndKnocking_1) |
    eventALeftEngineDeparted50AndNoKnocking
    c(eventALeftEngineDeparted50AndNoKnocking)
19. c(eventALeftEngineDeparted50AndNoKnocking) → ε
20. c(eventARightEngineDeparted50AndNoKnocking) → ε
21. c(eventInitialization) → eventAAdjustment c(eventAAdjustment)
22. c(eventALeftEngineApproached324_2) → eventALeftEngineDeparted162_1
    c(eventALeftEngineDeparted162_1)
23. c(eventARightEngineApproached324_2) → eventARightEngineDeparted162_1
    c(eventARightEngineDeparted162_1)
24. c(eventALeftEngineApproached50AndKnockingAndRef01Chosen) →
    eventALeftEngineDepartedRef01 c(eventALeftEngineDepartedRef01)
25. c(eventALeftEngineApproached50AndNoKnocking_8) → eventARightEngineAp-
    proached50AndNoKnocking_8 c(eventARightEngineApproached50AndNoKnocking_8) |
    eventARightEngi-neApproached50AndKnockingAndRef01Chosen
    c(eventARightEngineApproached50AndKnockingAndRef01Chosen) |
    eventARightEngineAp-proached50AndKnockingAndRef02Chosen
    c(eventARightEngineApproached50AndKnockingAndRef02Chosen) |
    eventARightEngineAp-proached50AndKnockingAndRef03Chosen
    c(eventARightEngineApproached50AndKnockingAndRef03Chosen) |
    eventARightEngineAp-proached50AndKnockingAndRef04Chosen
    c(eventARightEngineApproached50AndKnockingAndRef04Chosen) |
    eventARightEngineAp-proached50AndKnockingAndRef05Chosen
    c(eventARightEngineApproached50AndKnockingAndRef05Chosen) |
    eventARightEngineAp-proached50AndKnockingAndRef06Chosen
    c(eventARightEngineApproached50AndKnockingAndRef06Chosen) |
    eventARightEngineAp-proached50AndKnockingAndRef07Chosen
    c(eventARightEngineApproached50AndKnockingAndRef07Chosen) |
    eventARightEngineAp-proached50AndKnockingAndRef08Chosen
    c(eventARightEngineApproached50AndKnockingAndRef08Chosen) |
    eventARightEngineAp-proached50AndKnockingAndRef09Chosen
    c(eventARightEngineApproached50AndKnockingAndRef09Chosen) |
    eventARightEngineAp-proached50AndKnockingAndRef10Chosen
    c(eventARightEngineApproached50AndKnockingAndRef10Chosen)
26. c(eventARightEngineApproached50AndNoKnocking_8) → eventALeftEngineAp-
    proached50AndKnockingAndRef01Chosen

c(eventALeftEngineApproached50AndKnockingAndRef01Chosen) | eventALeftEngineApproached50AndNoKnocking_8 c(eventALeftEngineApproached50AndNoKnocking_8) | eventALeftEngine-Approached50AndKnockingAndRef02Chosen
c(eventALeftEngineApproached50AndKnockingAndRef02Chosen) | eventALeftEngineApproached50AndKnockingAndRef03Chosen
c(eventALeftEngineApproached50AndKnockingAndRef03Chosen) | eventALeftEngineApproached50AndKnockingAndRef04Chosen
c(eventALeftEngineApproached50AndKnockingAndRef04Chosen) | eventALeftEngineApproached50AndKnockingAndRef05Chosen
c(eventALeftEngineApproached50AndKnockingAndRef05Chosen) | eventALeftEngineApproached50AndKnockingAndRef06Chosen
c(eventALeftEngineApproached50AndKnockingAndRef06Chosen) | eventALeftEngineApproached50AndKnockingAndRef07Chosen
c(eventALeftEngineApproached50AndKnockingAndRef07Chosen) | eventALeftEngineApproached50AndKnockingAndRef08Chosen
c(eventALeftEngineApproached50AndKnockingAndRef08Chosen) | eventALeftEngineApproached50AndKnockingAndRef09Chosen
c(eventALeftEngineApproached50AndKnockingAndRef09Chosen) | eventALeftEngineApproached50AndKnockingAndRef10Chosen
c(eventALeftEngineApproached50AndKnockingAndRef10Chosen)

27. c(eventARightEngineDepartedRef01) → eventALeftEngineDepartedRef01AndNoKnocking c(eventALeftEngineDepartedRef01AndNoKnocking) | eventALeftEngineDepartedRef01AndKnocking c(eventALeftEngineDepartedRef01AndKnocking)
28. c(eventALeftEngineDepartedRef01AndNoKnocking) → ε
29. c(eventALeftEngineDepartedRef01AndKnocking) → eventARightEngineDeparted50AndKnocking_1 c(eventARightEngineDeparted50AndKnocking_1) | eventARightEngineDeparted50AndNoKnocking c(eventARightEngineDeparted50AndNoKnocking)
30. c(eventARightEngineApproached50AndKnockingAndRef01Chosen) → eventARightEngineDepartedRef01 c(eventARightEngineDepartedRef01)
31. c(eventALeftEngineDepartedRef02) → eventARightEngineDepartedRef02AndNoKnocking c(eventARightEngineDepartedRef02AndNoKnocking) | eventARightEngineDepartedRef02AndKnocking c(eventARightEngineDepartedRef02AndKnocking)
32. c(eventARightEngineDepartedRef02AndNoKnocking) → ε
33. c(eventARightEngineDepartedRef02AndKnocking) → eventARightEngineDeparted50AndKnocking_1 c(eventARightEngineDeparted50AndKnocking_1) | eventARightEngineDeparted50AndNoKnocking c(eventARightEngineDeparted50AndNoKnocking)
34. c(eventALeftEngineApproached50AndKnockingAndRef02Chosen) → eventALeftEngineDepartedRef02 c(eventALeftEngineDepartedRef02)
35. c(eventARightEngineDepartedRef02) → eventALeftEngineDepartedRef02AndNoKnocking c(eventALeftEngineDepartedRef02AndNoKnocking) | eventALeftEngineDepartedRef02AndKnocking c(eventALeftEngineDepartedRef02AndKnocking)

36. c(eventALeftEngineDepartedRef02AndNoKnocking) → ε
37. c(eventALeftEngineDepartedRef02AndKnocking) →
    eventARightEngineDeparted50AndKnocking_1
    c(eventARightEngineDeparted50AndKnocking_1) |
    eventARightEngineDeparted50AndNoKnocking
    c(eventARightEngineDeparted50AndNoKnocking)
38. c(eventARightEngineApproached50AndKnockingAndRef02Chosen) →
    eventARightEngineDepartedRef02 c(eventARightEngineDepartedRef02)
39. c(eventALeftEngineDepartedRef03) → eventARightEngineDepartedRef03AndNoKnocking
    c(eventARightEngineDepartedRef03AndNoKnocking) |
    eventARightEngineDepartedRef03AndKnocking
    c(eventARightEngineDepartedRef03AndKnocking)
40. c(eventARightEngineDepartedRef03AndNoKnocking) → ε
41. c(eventARightEngineDepartedRef03AndKnocking) → ε | eventARightEngineDe-
    parted50AndKnocking_1 c(eventARightEngineDeparted50AndKnocking_1) |
    eventARightEngineDe-parted50AndNoKnocking
    c(eventARightEngineDeparted50AndNoKnocking)
42. c(eventALeftEngineApproached50AndKnockingAndRef03Chosen) →
    eventALeftEngineDepartedRef03 c(eventALeftEngineDepartedRef03)
43. c(eventARightEngineDepartedRef03) → eventALeftEngineDepartedRef03AndNoKnocking
    c(eventALeftEngineDepartedRef03AndNoKnocking) |
    eventALeftEngineDepartedRef03AndKnocking
    c(eventALeftEngineDepartedRef03AndKnocking)
44. c(eventALeftEngineDepartedRef03AndNoKnocking) → ε
45. c(eventALeftEngineDepartedRef03AndKnocking) →
    eventARightEngineDeparted50AndKnocking_1
    c(eventARightEngineDeparted50AndKnocking_1) |
    eventARightEngineDeparted50AndNoKnocking
    c(eventARightEngineDeparted50AndNoKnocking)
46. c(eventARightEngineApproached50AndKnockingAndRef03Chosen) →
    eventARightEngineDepartedRef03 c(eventARightEngineDepartedRef03)
47. c(eventALeftEngineDepartedRef04) → eventARightEngineDepartedRef04AndNoKnocking
    c(eventARightEngineDepartedRef04AndNoKnocking) |
    eventARightEngineDepartedRef04AndKnocking
    c(eventARightEngineDepartedRef04AndKnocking)
48. c(eventARightEngineDepartedRef04AndNoKnocking) → ε
49. c(eventARightEngineDepartedRef04AndKnocking) →
    eventARightEngineDeparted50AndKnocking_1
    c(eventARightEngineDeparted50AndKnocking_1) |
    eventARightEngineDeparted50AndNoKnocking
    c(eventARightEngineDeparted50AndNoKnocking)
50. c(eventALeftEngineApproached50AndKnockingAndRef04Chosen) →
    eventALeftEngineDepartedRef04 c(eventALeftEngineDepartedRef04)
51. c(eventARightEngineDepartedRef04) → eventALeftEngineDepartedRef04AndNoKnocking
    c(eventALeftEngineDepartedRef04AndNoKnocking) |

eventALeftEngineDepartedRef04AndKnocking
c(eventALeftEngineDepartedRef04AndKnocking)

52. c(eventALeftEngineDepartedRef04AndNoKnocking) → ε
53. c(eventALeftEngineDepartedRef04AndKnocking) →
eventARightEngineDeparted50AndKnocking_1
c(eventARightEngineDeparted50AndKnocking_1) |
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking)
54. c(eventARightEngineApproached50AndKnockingAndRef04Chosen) →
eventARightEngineDepartedRef04 c(eventARightEngineDepartedRef04)
55. c(eventALeftEngineDepartedRef05) → eventARightEngineDepartedRef05AndNoKnocking
c(eventARightEngineDepartedRef05AndNoKnocking) |
eventARightEngineDepartedRef05AndKnocking
c(eventARightEngineDepartedRef05AndKnocking)
56. c(eventARightEngineDepartedRef05AndNoKnocking) → ε
57. c(eventARightEngineDepartedRef05AndKnocking) →
eventARightEngineDeparted50AndKnocking_1
c(eventARightEngineDeparted50AndKnocking_1) |
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking)
58. c(eventALeftEngineApproached50AndKnockingAndRef05Chosen) →
eventALeftEngineDepartedRef05 c(eventALeftEngineDepartedRef05)
59. c(eventARightEngineDepartedRef05) → eventALeftEngineDepartedRef05AndNoKnocking
c(eventALeftEngineDepartedRef05AndNoKnocking) |
eventALeftEngineDepartedRef05AndKnocking
c(eventALeftEngineDepartedRef05AndKnocking)
60. c(eventALeftEngineDepartedRef05AndNoKnocking) → ε
61. c(eventALeftEngineDepartedRef05AndKnocking) →
eventARightEngineDeparted50AndKnocking_1
c(eventARightEngineDeparted50AndKnocking_1) |
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking)
62. c(eventARightEngineApproached50AndKnockingAndRef05Chosen) →
eventARightEngineDepartedRef05 c(eventARightEngineDepartedRef05)
63. c(eventALeftEngineDepartedRef06) → eventARightEngineDepartedRef06AndNoKnocking
c(eventARightEngineDepartedRef06AndNoKnocking) |
eventARightEngineDepartedRef06AndKnocking
c(eventARightEngineDepartedRef06AndKnocking)
64. c(eventARightEngineDepartedRef06AndNoKnocking) → ε
65. c(eventARightEngineDepartedRef06AndKnocking) →
eventARightEngineDeparted50AndKnocking_1
c(eventARightEngineDeparted50AndKnocking_1) |
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking)
66. c(eventALeftEngineApproached50AndKnockingAndRef06Chosen) →
eventALeftEngineDepartedRef06 c(eventALeftEngineDepartedRef06)

67. c(eventARightEngineDepartedRef06) → eventALeftEngineDepartedRef06AndNoKnocking c(eventALeftEngineDepartedRef06AndNoKnocking) | eventALeftEngineDepartedRef06AndKnocking c(eventALeftEngineDepartedRef06AndKnocking)
68. c(eventALeftEngineDepartedRef06AndNoKnocking) → ε
69. c(eventALeftEngineDepartedRef06AndKnocking) → eventARightEngineDeparted50AndKnocking_1 c(eventARightEngineDeparted50AndKnocking_1) | eventARightEngineDeparted50AndNoKnocking c(eventARightEngineDeparted50AndNoKnocking)
70. c(eventARightEngineApproached50AndKnockingAndRef06Chosen) → eventARightEngineDepartedRef06 c(eventARightEngineDepartedRef06)
71. c(eventALeftEngineDepartedRef07) → eventARightEngineDepartedRef07AndNoKnocking c(eventARightEngineDepartedRef07AndNoKnocking) | eventARightEngineDepartedRef07AndKnocking c(eventARightEngineDepartedRef07AndKnocking)
72. c(eventARightEngineDepartedRef07AndNoKnocking) → ε
73. c(eventARightEngineDepartedRef07AndKnocking) → eventARightEngineDeparted50AndKnocking_1 c(eventARightEngineDeparted50AndKnocking_1) | eventARightEngineDeparted50AndNoKnocking c(eventARightEngineDeparted50AndNoKnocking)
74. c(eventALeftEngineApproached50AndKnockingAndRef07Chosen) → eventALeftEngineDepartedRef07 c(eventALeftEngineDepartedRef07)
75. c(eventARightEngineDepartedRef07) → eventALeftEngineDepartedRef07AndNoKnocking c(eventALeftEngineDepartedRef07AndNoKnocking) | eventALeftEngineDepartedRef07AndKnocking c(eventALeftEngineDepartedRef07AndKnocking)
76. c(eventALeftEngineDepartedRef07AndNoKnocking) → ε
77. c(eventALeftEngineDepartedRef07AndKnocking) → eventARightEngineDeparted50AndKnocking_1 c(eventARightEngineDeparted50AndKnocking_1) | eventARightEngineDeparted50AndNoKnocking c(eventARightEngineDeparted50AndNoKnocking)
78. c(eventARightEngineApproached50AndKnockingAndRef07Chosen) → eventARightEngineDepartedRef07 c(eventARightEngineDepartedRef07)
79. c(eventALeftEngineDepartedRef08) → eventARightEngineDepartedRef08AndNoKnocking c(eventARightEngineDepartedRef08AndNoKnocking) | eventARightEngineDepartedRef08AndKnocking c(eventARightEngineDepartedRef08AndKnocking)
80. c(eventARightEngineDepartedRef08AndNoKnocking) → ε
81. c(eventARightEngineDepartedRef08AndKnocking) → eventARightEngineDeparted50AndKnocking_1 c(eventARightEngineDeparted50AndKnocking_1) | eventARightEngineDeparted50AndNoKnocking c(eventARightEngineDeparted50AndNoKnocking)

82. c(eventALeftEngineApproached50AndKnockingAndRef08Chosen) →
eventALeftEngineDepartedRef08 c(eventALeftEngineDepartedRef08)
83. c(eventARightEngineDepartedRef08) → eventALeftEngineDepartedRef08AndNoKnocking
c(eventALeftEngineDepartedRef08AndNoKnocking) |
eventALeftEngineDepartedRef08AndKnocking
c(eventALeftEngineDepartedRef08AndKnocking)
84. c(eventALeftEngineDepartedRef08AndNoKnocking) → ε
85. c(eventALeftEngineDepartedRef08AndKnocking) →
eventARightEngineDeparted50AndKnocking_1
c(eventARightEngineDeparted50AndKnocking_1) |
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking)
86. c(eventARightEngineApproached50AndKnockingAndRef08Chosen) →
eventARightEngineDepartedRef08 c(eventARightEngineDepartedRef08)
87. c(eventALeftEngineDepartedRef09) → eventARightEngineDepartedRef09AndNoKnocking
c(eventARightEngineDepartedRef09AndNoKnocking) |
eventARightEngineDepartedRef09AndKnocking
c(eventARightEngineDepartedRef09AndKnocking)
88. c(eventARightEngineDepartedRef09AndNoKnocking) → ε
89. c(eventARightEngineDepartedRef09AndKnocking) →
eventARightEngineDeparted50AndKnocking_1
c(eventARightEngineDeparted50AndKnocking_1) |
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking)
90. c(eventALeftEngineApproached50AndKnockingAndRef09Chosen) →
eventALeftEngineDepartedRef09 c(eventALeftEngineDepartedRef09)
91. c(eventARightEngineDepartedRef09) → eventALeftEngineDepartedRef09AndNoKnocking
c(eventALeftEngineDepartedRef09AndNoKnocking) |
eventALeftEngineDepartedRef09AndKnocking
c(eventALeftEngineDepartedRef09AndKnocking)
92. c(eventALeftEngineDepartedRef09AndNoKnocking) → ε
93. c(eventALeftEngineDepartedRef09AndKnocking) →
eventARightEngineDeparted50AndKnocking_1
c(eventARightEngineDeparted50AndKnocking_1) |
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking)
94. c(eventARightEngineApproached50AndKnockingAndRef09Chosen) →
eventARightEngineDepartedRef09 c(eventARightEngineDepartedRef09)
95. c(eventALeftEngineDepartedRef10) → eventARightEngineDepartedRef10AndNoKnocking
c(eventARightEngineDepartedRef10AndNoKnocking) |
eventARightEngineDepartedRef10AndKnocking
c(eventARightEngineDepartedRef10AndKnocking)
96. c(eventARightEngineDepartedRef10AndNoKnocking) → ε
97. c(eventARightEngineDepartedRef10AndKnocking) →
eventARightEngineDeparted50AndKnocking_1
c(eventARightEngineDeparted50AndKnocking_1) |

eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking)

98. c(eventALeftEngineApproached50AndKnockingAndRef10Chosen) →
eventALeftEngineDepartedRef10 c(eventALeftEngineDepartedRef10)
99. c(eventARightEngineDepartedRef10) → eventALeftEngineDepartedRef10AndNoKnocking
c(eventALeftEngineDepartedRef10AndNoKnocking) |
eventALeftEngineDepartedRef10AndKnocking
c(eventALeftEngineDepartedRef10AndKnocking)
100. c(eventALeftEngineDepartedRef10AndNoKnocking) → ε
101. c(eventALeftEngineDepartedRef10AndKnocking) →
eventARightEngineDeparted50AndKnocking_1
c(eventARightEngineDeparted50AndKnocking_1) |
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking)
102. c(eventARightEngineApproached50AndKnockingAndRef10Chosen) →
eventARightEngineDepartedRef10 c(eventARightEngineDepartedRef10)
103. c(eventALeftEngineApproached324_3) → eventALeftEngineApproached324_4
c(eventALeftEngineApproached324_4)
104. c(eventARightEngineApproached324_3) → eventARightEngineApproached324_4
c(eventARightEngineApproached324_4)
105. c(eventALeftEngineDeparted162_2) → eventARightEngineApproached324_5
c(eventARightEngineApproached324_5)
106. c(eventARightEngineDeparted162_2) → eventALeftEngineApproached324_5
c(eventALeftEngineApproached324_5)
107. c(eventALeftEngineApproached324_4) → eventALeftEngineDeparted162_2
c(eventALeftEngineDeparted162_2)
108. c(eventARightEngineApproached324_4) → eventARightEngineDeparted162_2
c(eventARightEngineDeparted162_2)
109. c(eventALeftEngineApproached324AndKnocking_1) → eventALeftEngineDeparted375_1
c(eventALeftEngineDeparted375_1)
110. c(eventALeftEngineApproached324_5) → eventALeftEngineApproached324AndKnocking_1
c(eventALeftEngineApproached324AndKnocking_1) | eventALeftEngineApproached324_6
c(eventALeftEngineApproached324_6)
111. c(eventARightEngineApproached324_5) → eventARightEngineApproached324_6
c(eventARightEngineApproached324_6) |
eventARightEngineApproached324AndKnocking_1
c(eventARightEngineApproached324AndKnocking_1)
112. c(eventALeftEngineDeparted375_1) → eventARightEngineApproached324_25
c(eventARightEngineApproached324_25)
113. c(eventALeftEngineDeparted162_3) → eventARightEngineApproached324_7
c(eventARightEngineApproached324_7)
114. c(eventARightEngineDeparted162_3) → eventALeftEngineApproached324_7
c(eventALeftEngineApproached324_7)
115. c(eventALeftEngineApproached324_6) → eventALeftEngineDeparted162_3
c(eventALeftEngineDeparted162_3)

116.c(eventARightEngineApproached324_6) → eventARightEngineDeparted162_3
c(eventARightEngineDeparted162_3)

117.c(eventARightEngineApproached324AndKnocking_1) → eventARightEngineDeparted375_1
c(eventARightEngineDeparted375_1)

118.c(eventARightEngineDeparted375_1) → eventALeftEngineApproached324_25
c(eventALeftEngineApproached324_25)

119.c(eventALeftEngineApproached324AndKnocking_7) → eventALeftEngineDeparted375_7
c(eventALeftEngineDeparted375_7)

120.c(eventALeftEngineApproached324_17) →
eventALeftEngineApproached324AndKnocking_7
c(eventALeftEngineApproached324AndKnocking_7) | eventALeftEngineApproached324_18
c(eventALeftEngineApproached324_18)

121.c(eventARightEngineApproached324_17) → eventARightEngineApproached324_18
c(eventARightEngineApproached324_18) |
eventARightEngineApproached324AndKnocking_7
c(eventARightEngineApproached324AndKnocking_7)

122.c(eventALeftEngineDeparted375_7) → eventARightEngineApproached324_31
c(eventARightEngineApproached324_31)

123.c(eventALeftEngineDeparted162_9) → eventARightEngineApproached324_19
c(eventARightEngineApproached324_19)

124.c(eventARightEngineDeparted162_9) → eventALeftEngineApproached324_19
c(eventALeftEngineApproached324_19)

125.c(eventALeftEngineApproached324_18) → eventALeftEngineDeparted162_9
c(eventALeftEngineDeparted162_9)

126.c(eventARightEngineApproached324_18) → eventARightEngineDeparted162_9
c(eventARightEngineDeparted162_9)

127.c(eventARightEngineApproached324AndKnocking_7) → eventARightEngineDeparted375_7
c(eventARightEngineDeparted375_7)

128.c(eventARightEngineDeparted375_7) → eventALeftEngineApproached324_31
c(eventALeftEngineApproached324_31)

129.c(eventALeftEngineApproached324AndKnocking_12) → eventALeftEngineDeparted375_12
c(eventALeftEngineDeparted375_12)

130.c(eventALeftEngineApproached324_35) →
eventALeftEngineApproached324AndKnocking_12
c(eventALeftEngineApproached324AndKnocking_12) |
eventALeftEngineApproached324_36 c(eventALeftEngineApproached324_36)

131.c(eventARightEngineApproached324_35) → eventARightEngineApproached324_36
c(eventARightEngineApproached324_36) |
eventARightEngineApproached324AndKnocking_12
c(eventARightEngineApproached324AndKnocking_12)

132.c(eventALeftEngineDeparted375_12) → eventALeftEngineDeparted200_4
c(eventALeftEngineDeparted200_4)

133.c(eventALeftEngineDeparted162_18) → eventALeftEngineApproached324_35
c(eventALeftEngineApproached324_35)

134.c(eventARightEngineDeparted162_18) → eventARightEngineApproached324_35
c(eventARightEngineApproached324_35)

135.c(eventALeftEngineApproached324_36) → eventALeftEngineDeparted162_18
c(eventALeftEngineDeparted162_18)

136.c(eventARightEngineApproached324_36) → eventARightEngineDeparted162_18
c(eventARightEngineDeparted162_18)

137.c(eventARightEngineApproached324AndKnocking_12) →
eventARightEngineDeparted375_12 c(eventARightEngineDeparted375_12)

138.c(eventARightEngineDeparted375_12) → eventARightEngineDeparted200_6
c(eventARightEngineDeparted200_6)

139.c(eventALeftEngineApproached50AndNoKnocking_1) → eventARightEngineAp-
proached50AndNoKnocking_1 c(eventARightEngineApproached50AndNoKnocking_1)

140.c(eventALeftEngineApproached324_7) → eventALeftEngineApproached324_8
c(eventALeftEngineApproached324_8) | eventALeftEngineApproached324AndKnocking_2
c(eventALeftEngineApproached324AndKnocking_2)

141.c(eventARightEngineApproached324_7) → eventARightEngineApproached324_8
c(eventARightEngineApproached324_8) |
eventARightEngineApproached324AndKnocking_2
c(eventARightEngineApproached324AndKnocking_2)

142.c(eventALeftEngineDeparted162_4) → eventARightEngineApproached324_9
c(eventARightEngineApproached324_9)

143.c(eventARightEngineDeparted162_4) → eventALeftEngineApproached324_9
c(eventALeftEngineApproached324_9)

144.c(eventALeftEngineApproached324_8) → eventALeftEngineDeparted162_4
c(eventALeftEngineDeparted162_4)

145.c(eventARightEngineApproached324_8) → eventARightEngineDeparted162_4
c(eventARightEngineDeparted162_4)

146.c(eventALeftEngineApproached324_9) → eventALeftEngineApproached324_10
c(eventALeftEngineApproached324_10) | eventALeftEngineApproached324AndKnocking_3
c(eventALeftEngineApproached324AndKnocking_3)

147.c(eventARightEngineApproached324_9) → eventARightEngineApproached324_10
c(eventARightEngineApproached324_10) |
eventARightEngineApproached324AndKnocking_3
c(eventARightEngineApproached324AndKnocking_3)

148.c(eventALeftEngineDeparted162_5) → eventARightEngineApproached324_11
c(eventARightEngineApproached324_11)

149.c(eventARightEngineDeparted162_5) → eventALeftEngineApproached324_11
c(eventALeftEngineApproached324_11)

150.c(eventALeftEngineApproached324_10) → eventALeftEngineDeparted162_5
c(eventALeftEngineDeparted162_5)

151.c(eventARightEngineApproached324_10) → eventARightEngineDeparted162_5
c(eventARightEngineDeparted162_5)

152.c(eventALeftEngineApproached324_11) → eventALeftEngineApproached324_12
c(eventALeftEngineApproached324_12) | eventALeftEngineApproached324AndKnocking_4
c(eventALeftEngineApproached324AndKnocking_4)

153.c(eventARightEngineApproached324_11) → eventARightEngineApproached324_12
c(eventARightEngineApproached324_12) |

eventARightEngineApproached324AndKnocking_4
c(eventARightEngineApproached324AndKnocking_4)

154.c(eventALeftEngineDeparted162_6) → eventARightEngineApproached324_13
c(eventARightEngineApproached324_13)

155.c(eventARightEngineDeparted162_6) → eventALeftEngineApproached324_13
c(eventALeftEngineApproached324_13)

156.c(eventALeftEngineApproached324_12) → eventALeftEngineDeparted162_6
c(eventALeftEngineDeparted162_6)

157.c(eventARightEngineApproached324_12) → eventARightEngineDeparted162_6
c(eventARightEngineDeparted162_6)

158.c(eventALeftEngineApproached324_13) → eventALeftEngineApproached324_14
c(eventALeftEngineApproached324_14) | eventALeftEngineApproached324AndKnocking_5
c(eventALeftEngineApproached324AndKnocking_5)

159.c(eventARightEngineApproached324_13) → eventARightEngineApproached324_14
c(eventARightEngineApproached324_14) |
eventARightEngineApproached324AndKnocking_5
c(eventARightEngineApproached324AndKnocking_5)

160.c(eventALeftEngineDeparted162_7) → eventARightEngineApproached324_15
c(eventARightEngineApproached324_15)

161.c(eventARightEngineDeparted162_7) → eventALeftEngineApproached324_15
c(eventALeftEngineApproached324_15)

162.c(eventALeftEngineApproached324_14) → eventALeftEngineDeparted162_7
c(eventALeftEngineDeparted162_7)

163.c(eventARightEngineApproached324_14) → eventARightEngineDeparted162_7
c(eventARightEngineDeparted162_7)

164.c(eventARightEngineApproached324AndKnocking_2) → eventARightEngineDeparted375_2
c(eventARightEngineDeparted375_2)

165.c(eventARightEngineDeparted375_2) → eventALeftEngineApproached324_27
c(eventALeftEngineApproached324_27)

166.c(eventARightEngineApproached324AndKnocking_3) → eventARightEngineDeparted375_3
c(eventARightEngineDeparted375_3)

167.c(eventARightEngineDeparted375_3) → eventALeftEngineApproached324_27
c(eventALeftEngineApproached324_27)

168.c(eventARightEngineApproached324AndKnocking_4) → eventARightEngineDeparted375_4
c(eventARightEngineDeparted375_4)

169.c(eventARightEngineDeparted375_4) → eventALeftEngineApproached324_29
c(eventALeftEngineApproached324_29)

170.c(eventARightEngineApproached324AndKnocking_5) → eventARightEngineDeparted375_5
c(eventARightEngineDeparted375_5)

171.c(eventARightEngineDeparted375_5) → eventALeftEngineApproached324_29
c(eventALeftEngineApproached324_29)

172.c(eventALeftEngineApproached324_23) → eventALeftEngineApproached324_24
c(eventALeftEngineApproached324_24) |
eventALeftEngineApproached324AndKnocking_10
c(eventALeftEngineApproached324AndKnocking_10)

173.c(eventARightEngineApproached324_23) → eventARightEngineApproached324_24
c(eventARightEngineApproached324_24) |
eventARightEngineApproached324AndKnocking_10
c(eventARightEngineApproached324AndKnocking_10)

174.c(eventALeftEngineDeparted162_12) → eventARightEngineApproached324_23
c(eventARightEngineApproached324_23)

175.c(eventARightEngineDeparted162_12) → eventALeftEngineApproached324_23
c(eventALeftEngineApproached324_23)

176.c(eventALeftEngineApproached324_24) → eventALeftEngineDeparted162_12
c(eventALeftEngineDeparted162_12)

177.c(eventARightEngineApproached324_24) → eventARightEngineDeparted162_12
c(eventARightEngineDeparted162_12)

178.c(eventARightEngineApproached324AndKnocking_10) →
eventARightEngineDeparted375_10 c(eventARightEngineDeparted375_10)

179.c(eventARightEngineDeparted375_10) → eventALeftEngineApproached324_35
c(eventALeftEngineApproached324_35)

180.c(eventALeftEngineApproached324AndKnocking_10) → eventALeftEngineDeparted375_10
c(eventALeftEngineDeparted375_10)

181.c(eventALeftEngineDeparted375_10) → eventARightEngineApproached324_35
c(eventARightEngineApproached324_35)

182.c(eventALeftEngineApproached324AndKnocking_5) → eventALeftEngineDeparted375_5
c(eventALeftEngineDeparted375_5)

183.c(eventALeftEngineDeparted375_5) → eventARightEngineApproached324_29
c(eventARightEngineApproached324_29)

184.c(eventALeftEngineApproached324AndKnocking_4) → eventALeftEngineDeparted375_4
c(eventALeftEngineDeparted375_4)

185.c(eventALeftEngineDeparted375_4) → eventARightEngineApproached324_29
c(eventARightEngineApproached324_29)

186.c(eventALeftEngineApproached324AndKnocking_3) → eventALeftEngineDeparted375_3
c(eventALeftEngineDeparted375_3)

187.c(eventALeftEngineDeparted375_3) → eventARightEngineApproached324_27
c(eventARightEngineApproached324_27)

188.c(eventALeftEngineApproached324AndKnocking_2) → eventALeftEngineDeparted375_2
c(eventALeftEngineDeparted375_2)

189.c(eventALeftEngineDeparted375_2) → eventARightEngineApproached324_27
c(eventARightEngineApproached324_27)

190.c(eventALeftEngineApproached324_15) → eventALeftEngineApproached324_16
c(eventALeftEngineApproached324_16) | eventALeftEngineApproached324AndKnocking_6
c(eventALeftEngineApproached324AndKnocking_6)

191.c(eventARightEngineApproached324_15) → eventARightEngineApproached324_16
c(eventARightEngineApproached324_16) |
eventARightEngineApproached324AndKnocking_6
c(eventARightEngineApproached324AndKnocking_6)

192.c(eventALeftEngineDeparted162_8) → eventARightEngineApproached324_17
c(eventARightEngineApproached324_17)

193.c(eventARightEngineDeparted162_8) → eventALeftEngineApproached324_17
c(eventALeftEngineApproached324_17)

194.c(eventALeftEngineApproached324_16) → eventALeftEngineDeparted162_8
c(eventALeftEngineDeparted162_8)

195.c(eventARightEngineApproached324_16) → eventARightEngineDeparted162_8
c(eventARightEngineDeparted162_8)

196.c(eventARightEngineApproached324AndKnocking_6) → eventARightEngineDeparted375_6
c(eventARightEngineDeparted375_6)

197.c(eventARightEngineDeparted375_6) → eventALeftEngineApproached324_31
c(eventALeftEngineApproached324_31)

198.c(eventALeftEngineApproached324AndKnocking_6) → eventALeftEngineDeparted375_6
c(eventALeftEngineDeparted375_6)

199.c(eventALeftEngineDeparted375_6) → eventARightEngineApproached324_31
c(eventARightEngineApproached324_31)

200.c(eventALeftEngineDeparted200_4) → eventARightEngineDeparted200_3
c(eventARightEngineDeparted200_3)

201.c(eventARightEngineDeparted200_3) → eventALeftEngineDeparted200_5
c(eventALeftEngineDeparted200_5)

202.c(eventALeftEngineDeparted200_5) → eventARightEngineDeparted200_4
c(eventARightEngineDeparted200_4)

203.c(eventARightEngineDeparted200_4) → eventALeftEngineDeparted200_6
c(eventALeftEngineDeparted200_6)

204.c(eventALeftEngineDeparted200_6) → eventARightEngineDeparted200_5
c(eventARightEngineDeparted200_5)

205.c(eventALeftEngineApproached324_37) → eventALeftEngineApproached324_38
c(eventALeftEngineApproached324_38)

206.c(eventARightEngineApproached324_37) → eventARightEngineApproached324_38
c(eventARightEngineApproached324_38)

207.c(eventALeftEngineDeparted162_19) → eventARightEngineApproached324_39
c(eventARightEngineApproached324_39)

208.c(eventARightEngineDeparted162_19) → eventALeftEngineApproached324_39
c(eventALeftEngineApproached324_39)

209.c(eventALeftEngineApproached324_38) → eventALeftEngineDeparted162_19
c(eventALeftEngineDeparted162_19)

210.c(eventARightEngineApproached324_38) → eventARightEngineDeparted162_19
c(eventARightEngineDeparted162_19)

211.c(eventALeftEngineApproached324_39) → eventALeftEngineApproached324_40
c(eventALeftEngineApproached324_40)

212.c(eventARightEngineApproached324_39) → eventARightEngineApproached324_40
c(eventARightEngineApproached324_40)

213.c(eventALeftEngineDeparted162_20) → eventARightEngineApproached324_41
c(eventARightEngineApproached324_41)

214.c(eventARightEngineDeparted162_20) → eventALeftEngineApproached324_41
c(eventALeftEngineApproached324_41)

215.c(eventALeftEngineApproached324_40) → eventALeftEngineDeparted162_20
c(eventALeftEngineDeparted162_20)

216.c(eventARightEngineApproached324_40) → eventARightEngineDeparted162_20
c(eventARightEngineDeparted162_20)

217.c(eventALeftEngineApproached324AndKnocking_13) → eventALeftEngineDeparted375_13
c(eventALeftEngineDeparted375_13)

218.c(eventALeftEngineApproached324_41) →
eventALeftEngineApproached324AndKnocking_13
c(eventALeftEngineApproached324AndKnocking_13) |
eventALeftEngineApproached324_42 c(eventALeftEngineApproached324_42)

219.c(eventARightEngineApproached324_41) → eventARightEngineApproached324_42
c(eventARightEngineApproached324_42) |
eventARightEngineApproached324AndKnocking_13
c(eventARightEngineApproached324AndKnocking_13)

220.c(eventALeftEngineDeparted375_13) → eventARightEngineApproached324_43
c(eventARightEngineApproached324_43)

221.c(eventALeftEngineDeparted162_21) → eventARightEngineApproached324_41
c(eventARightEngineApproached324_41)

222.c(eventARightEngineDeparted162_21) → eventALeftEngineApproached324_41
c(eventALeftEngineApproached324_41)

223.c(eventALeftEngineApproached324_42) → eventALeftEngineDeparted162_21
c(eventALeftEngineDeparted162_21)

224.c(eventARightEngineApproached324_42) → eventARightEngineDeparted162_21
c(eventARightEngineDeparted162_21)

225.c(eventARightEngineApproached324AndKnocking_13) →
eventARightEngineDeparted375_13 c(eventARightEngineDeparted375_13)

226.c(eventARightEngineDeparted375_13) → eventALeftEngineApproached324_43
c(eventALeftEngineApproached324_43)

227.c(eventALeftEngineApproached324_43) → eventALeftEngineApproached324_44
c(eventALeftEngineApproached324_44) |
eventALeftEngineApproached324AndKnocking_11
c(eventALeftEngineApproached324AndKnocking_11)

228.c(eventARightEngineApproached324_43) → eventARightEngineApproached324_44
c(eventARightEngineApproached324_44) |
eventARightEngineApproached324AndKnocking_11
c(eventARightEngineApproached324AndKnocking_11)

229.c(eventALeftEngineDeparted162_22) → eventALeftEngineApproached324_43
c(eventALeftEngineApproached324_43)

230.c(eventARightEngineDeparted162_22) → eventARightEngineApproached324_43
c(eventARightEngineApproached324_43)

231.c(eventALeftEngineApproached324_44) → eventALeftEngineDeparted162_22
c(eventALeftEngineDeparted162_22)

232.c(eventARightEngineApproached324_44) → eventARightEngineDeparted162_22
c(eventARightEngineDeparted162_22)

233.c(eventALeftEngineDeparted200_7) → eventARightEngineDeparted200_7
c(eventARightEngineDeparted200_7)

234.c(eventARightEngineDeparted200_7) → eventALeftEngineDeparted200_8
c(eventALeftEngineDeparted200_8)

235.c(eventALeftEngineDeparted200_8) → eventARightEngineDeparted200_8
c(eventARightEngineDeparted200_8)

236.c(eventARightEngineDeparted200_8) → eventALeftEngineDeparted200_9
c(eventALeftEngineDeparted200_9)

237.c(eventALeftEngineDeparted200_9) → eventARightEngineApproached324_37
c(eventARightEngineApproached324_37)

238.c(eventALeftEngineDeparted50AndKnocking_2) →
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking) |
eventARightEngineDeparted50AndKnocking_3
c(eventARightEngineDeparted50AndKnocking_3)

239.c(eventARightEngineDeparted50AndKnocking_2) →
eventALeftEngineDeparted50AndNoKnocking
c(eventALeftEngineDeparted50AndNoKnocking) |
eventALeftEngineDeparted50AndKnocking_2
c(eventALeftEngineDeparted50AndKnocking_2)

240.c(eventALeftEngineDeparted50AndKnocking_3) →
eventARightEngineDeparted50AndNoKnocking
c(eventARightEngineDeparted50AndNoKnocking) |
eventARightEngineDeparted50AndKnocking_4
c(eventARightEngineDeparted50AndKnocking_4)

241.c(eventARightEngineDeparted50AndKnocking_3) →
eventALeftEngineDeparted50AndNoKnocking
c(eventALeftEngineDeparted50AndNoKnocking) |
eventALeftEngineDeparted50AndKnocking_3
c(eventALeftEngineDeparted50AndKnocking_3)

242.c(eventARightEngineDeparted50AndKnocking_4) → ε

243.c(eventARightEngineDeparted200_5) → eventALeftEngineApproached324_37
c(eventALeftEngineApproached324_37)

244.c(eventARightEngineDeparted200_6) → eventALeftEngineDeparted200_7
c(eventALeftEngineDeparted200_7)

245.c(eventALeftEngineApproached324_29) → eventALeftEngineApproached324_30
c(eventALeftEngineApproached324_30) |
eventALeftEngineApproached324AndKnocking_11
c(eventALeftEngineApproached324AndKnocking_11)

246.c(eventARightEngineApproached324_29) → eventARightEngineApproached324_30
c(eventARightEngineApproached324_30) |
eventARightEngineApproached324AndKnocking_11
c(eventARightEngineApproached324AndKnocking_11)

247.c(eventALeftEngineDeparted162_15) → eventALeftEngineApproached324_31
c(eventALeftEngineApproached324_31)

248.c(eventARightEngineDeparted162_15) → eventARightEngineApproached324_31
c(eventARightEngineApproached324_31)

249.c(eventALeftEngineApproached324_30) → eventALeftEngineDeparted162_15
c(eventALeftEngineDeparted162_15)

250.c(eventARightEngineApproached324_30) → eventARightEngineDeparted162_15
c(eventARightEngineDeparted162_15)

251.c(eventALeftEngineApproached324_31) → eventALeftEngineApproached324_32
c(eventALeftEngineApproached324_32) |
eventALeftEngineApproached324AndKnocking_11
c(eventALeftEngineApproached324AndKnocking_11)

252.c(eventARightEngineApproached324_31) → eventARightEngineApproached324_32
c(eventARightEngineApproached324_32) |
eventARightEngineApproached324AndKnocking_11
c(eventARightEngineApproached324AndKnocking_11)

253.c(eventALeftEngineDeparted162_16) → eventALeftEngineApproached324_33
c(eventALeftEngineApproached324_33)

254.c(eventARightEngineDeparted162_16) → eventARightEngineApproached324_33
c(eventARightEngineApproached324_33)

255.c(eventALeftEngineApproached324_32) → eventALeftEngineDeparted162_16
c(eventALeftEngineDeparted162_16)

256.c(eventARightEngineApproached324_32) → eventARightEngineDeparted162_16
c(eventARightEngineDeparted162_16)

257.c(eventALeftEngineApproached324_33) → eventALeftEngineApproached324_34
c(eventALeftEngineApproached324_34) |
eventALeftEngineApproached324AndKnocking_11
c(eventALeftEngineApproached324AndKnocking_11)

258.c(eventARightEngineApproached324_33) → eventARightEngineApproached324_34
c(eventARightEngineApproached324_34) |
eventARightEngineApproached324AndKnocking_11
c(eventARightEngineApproached324AndKnocking_11)

259.c(eventALeftEngineDeparted162_17) → eventALeftEngineApproached324_35
c(eventALeftEngineApproached324_35)

260.c(eventARightEngineDeparted162_17) → eventARightEngineApproached324_35
c(eventARightEngineApproached324_35)

261.c(eventALeftEngineApproached324_34) → eventALeftEngineDeparted162_17
c(eventALeftEngineDeparted162_17)

262.c(eventARightEngineApproached324_34) → eventARightEngineDeparted162_17
c(eventARightEngineDeparted162_17)

263.c(eventARightEngineApproached324AndKnocking_11) →
eventARightEngineDeparted375_11 c(eventARightEngineDeparted375_11)

264.c(eventARightEngineDeparted375_11) →
eventALeftEngineApproached50AndNoKnocking_1
c(eventALeftEngineApproached50AndNoKnocking_1)

265.c(eventALeftEngineApproached324AndKnocking_11) → eventALeftEngineDeparted375_11
c(eventALeftEngineDeparted375_11)

266.c(eventALeftEngineDeparted375_11) → eventALeftEngineApproached50AndNoKnocking_1
c(eventALeftEngineApproached50AndNoKnocking_1)

267.c(eventALeftEngineApproached324AndKnocking_9) → eventALeftEngineDeparted375_9
c(eventALeftEngineDeparted375_9)

268.c(eventALeftEngineApproached324_21) →
eventALeftEngineApproached324AndKnocking_9
c(eventALeftEngineApproached324AndKnocking_9) | eventALeftEngineApproached324_22
c(eventALeftEngineApproached324_22)

269.c(eventARightEngineApproached324_21) → eventARightEngineApproached324_22
c(eventARightEngineApproached324_22) |
eventARightEngineApproached324AndKnocking_9
c(eventARightEngineApproached324AndKnocking_9)

270.c(eventALeftEngineDeparted375_9) → eventARightEngineApproached324_33
c(eventARightEngineApproached324_33)

271.c(eventALeftEngineDeparted162_11) → eventARightEngineApproached324_23
c(eventARightEngineApproached324_23)

272.c(eventARightEngineDeparted162_11) → eventALeftEngineApproached324_23
c(eventALeftEngineApproached324_23)

273.c(eventALeftEngineApproached324_22) → eventALeftEngineDeparted162_11
c(eventALeftEngineDeparted162_11)

274.c(eventARightEngineApproached324_22) → eventARightEngineDeparted162_11
c(eventARightEngineDeparted162_11)

275.c(eventARightEngineApproached324AndKnocking_9) → eventARightEngineDeparted375_9
c(eventARightEngineDeparted375_9)

276.c(eventARightEngineDeparted375_9) → eventALeftEngineApproached324_33
c(eventALeftEngineApproached324_33)

277.c(eventALeftEngineApproached324AndKnocking_8) → eventALeftEngineDeparted375_8
c(eventALeftEngineDeparted375_8)

278.c(eventALeftEngineApproached324_19) →
eventALeftEngineApproached324AndKnocking_8
c(eventALeftEngineApproached324AndKnocking_8) | eventALeftEngineApproached324_20
c(eventALeftEngineApproached324_20)

279.c(eventARightEngineApproached324_19) → eventARightEngineApproached324_20
c(eventARightEngineApproached324_20) |
eventARightEngineApproached324AndKnocking_8
c(eventARightEngineApproached324AndKnocking_8)

280.c(eventALeftEngineDeparted375_8) → eventARightEngineApproached324_33
c(eventARightEngineApproached324_33)

281.c(eventALeftEngineDeparted162_10) → eventARightEngineApproached324_21
c(eventARightEngineApproached324_21)

282.c(eventARightEngineDeparted162_10) → eventALeftEngineApproached324_21
c(eventALeftEngineApproached324_21)

283.c(eventALeftEngineApproached324_20) → eventALeftEngineDeparted162_10
c(eventALeftEngineDeparted162_10)

284.c(eventARightEngineApproached324_20) → eventARightEngineDeparted162_10
c(eventARightEngineDeparted162_10)

285.c(eventARightEngineApproached324AndKnocking_8) → eventARightEngineDeparted375_8
c(eventARightEngineDeparted375_8)

286.c(eventARightEngineDeparted375_8) → eventALeftEngineApproached324_33
c(eventALeftEngineApproached324_33)

287.c(eventALeftEngineApproached324_25) →
eventALeftEngineApproached324AndKnocking_11
c(eventALeftEngineApproached324AndKnocking_11) |
eventALeftEngineApproached324_26 c(eventALeftEngineApproached324_26)

288.c(eventARightEngineApproached324_25) →
eventARightEngineApproached324AndKnocking_11
c(eventARightEngineApproached324AndKnocking_11) |
eventARightEngineApproached324_26 c(eventARightEngineApproached324_26)

289.c(eventALeftEngineDeparted162_13) → eventALeftEngineApproached324_27
c(eventALeftEngineApproached324_27)

290.c(eventARightEngineDeparted162_13) → eventARightEngineApproached324_27
c(eventARightEngineApproached324_27)

291.c(eventALeftEngineApproached324_26) → eventALeftEngineDeparted162_13
c(eventALeftEngineDeparted162_13)

292.c(eventARightEngineApproached324_26) → eventARightEngineDeparted162_13
c(eventARightEngineDeparted162_13)

293.c(eventALeftEngineApproached324_27) →
eventALeftEngineApproached324AndKnocking_11
c(eventALeftEngineApproached324AndKnocking_11) |
eventALeftEngineApproached324_28 c(eventALeftEngineApproached324_28)

294.c(eventARightEngineApproached324_27) →
eventARightEngineApproached324AndKnocking_11
c(eventARightEngineApproached324AndKnocking_11) |
eventARightEngineApproached324_28 c(eventARightEngineApproached324_28)

295.c(eventALeftEngineDeparted162_14) → eventALeftEngineApproached324_29
c(eventALeftEngineApproached324_29)

296.c(eventARightEngineDeparted162_14) → eventARightEngineApproached324_29
c(eventARightEngineApproached324_29)

297.c(eventALeftEngineApproached324_28) → eventALeftEngineDeparted162_14
c(eventALeftEngineDeparted162_14)

298.c(eventARightEngineApproached324_28) → eventARightEngineDeparted162_14
c(eventARightEngineDeparted162_14)

299.c(eventALeftEngineDeparted200_1) → eventARightEngineDeparted200_1
c(eventARightEngineDeparted200_1)

300.c(eventARightEngineDeparted200_1) → eventALeftEngineDeparted200_2
c(eventALeftEngineDeparted200_2)

301.c(eventALeftEngineDeparted200_2) → eventARightEngineDeparted200_2
c(eventARightEngineDeparted200_2)

302.c(eventARightEngineDeparted200_2) → eventALeftEngineDeparted200_3
c(eventALeftEngineDeparted200_3)

303.c(eventARightEngineApproached50AndNoKnocking_1) → eventALeftEngineAp-
proached50AndNoKnocking_2 c(eventALeftEngineApproached50AndNoKnocking_2)

304.c(eventALeftEngineApproached50AndNoKnocking_2) → eventARightEngineAp-
proached50AndNoKnocking_2 c(eventARightEngineApproached50AndNoKnocking_2)

305.c(eventARightEngineApproached50AndNoKnocking_2) → eventALeftEngineAp-
proached50AndNoKnocking_3 c(eventALeftEngineApproached50AndNoKnocking_3)

306. c(eventALeftEngineApproached50AndNoKnocking_3) → eventARightEngineApproached50AndNoKnocking_3 c(eventARightEngineApproached50AndNoKnocking_3)
307. c(eventARightEngineApproached50AndNoKnocking_3) → eventALeftEngineApproached50AndNoKnocking_4 c(eventALeftEngineApproached50AndNoKnocking_4)
308. c(eventALeftEngineApproached50AndNoKnocking_4) → eventARightEngineApproached50AndNoKnocking_4 c(eventARightEngineApproached50AndNoKnocking_4)
309. c(eventARightEngineApproached50AndNoKnocking_4) → eventALeftEngineApproached50AndNoKnocking_5 c(eventALeftEngineApproached50AndNoKnocking_5)
310. c(eventALeftEngineApproached50AndNoKnocking_5) → eventARightEngineApproached50AndNoKnocking_5 c(eventARightEngineApproached50AndNoKnocking_5)
311. c(eventARightEngineApproached50AndNoKnocking_5) → eventALeftEngineApproached50AndNoKnocking_6 c(eventALeftEngineApproached50AndNoKnocking_6)
312. c(eventALeftEngineApproached50AndNoKnocking_6) → eventARightEngineApproached50AndNoKnocking_6 c(eventARightEngineApproached50AndNoKnocking_6)
313. c(eventARightEngineApproached50AndNoKnocking_6) → eventALeftEngineApproached50AndNoKnocking_7 c(eventALeftEngineApproached50AndNoKnocking_7)
314. c(eventALeftEngineApproached50AndNoKnocking_7) → eventARightEngineApproached50AndNoKnocking_7 c(eventARightEngineApproached50AndNoKnocking_7)
315. c(eventARightEngineApproached50AndNoKnocking_7) → eventALeftEngineApproached50AndNoKnocking_8 c(eventALeftEngineApproached50AndNoKnocking_8)

A.2 Productions of Specials 1-Reg Model

1. S → [___0 c([___0)
2. c([___0) → OpenSpecials_0___89 c(OpenSpecials_0___89)
3. c(Add_C_2+___1) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) |]___19 c(]___19) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28)
4. c(Add_I_2+___2) → Add_I_2+___2 c(Add_I_2+___2) | Edit_2+___6 c(Edit_2+___6) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_I_W_2+___26 c(Delete_I_W_2+___26)
5. c(DataC_N_2+___3) → Add_C_2+___1 c(Add_C_2+___1) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Delete_C_N_2+___8 c(Delete_C_N_2+___8) | DataE_N_2+___24 c(DataE_N_2+___24)
6. c(DataI_N_2+___4) → Add_I_2+___2 c(Add_I_2+___2) | DataC_N_2+___3 c(DataC_N_2+___3) | De-lete_I_N_2+___5 c(Delete_I_N_2+___5) | Edit_2+___6 c(Edit_2+___6) | DataE_N_2+___24 c(DataE_N_2+___24)
7. c(Delete_I_N_2+___5) → DCancel_I_N_2+___7 c(DCancel_I_N_2+___7) | DOK_2+_1___20 c(DOK_2+_1___20)
8. c(Edit_2+___6) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | Delete_C_E_N_2+___15 c(Delete_C_E_N_2+___15) | DataE_E_N_2+___36 c(DataE_E_N_2+___36)

9. c(DCancel_I_N_2+___7) → Add_I_2+___2 c(Add_I_2+___2) | DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Delete_I_N_2+___5 c(Delete_I_N_2+___5) | Edit_2+___6 c(Edit_2+___6) | DataE_N_2+___24 c(DataE_N_2+___24)
10. c(Delete_C_N_2+___8) → DCancel_C_N_2+___9 c(DCancel_C_N_2+___9) | DOK_2+_1___20 c(DOK_2+_1___20)
11. c(DCancel_C_N_2+___9) → Add_C_2+___1 c(Add_C_2+___1) | DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Delete_C_N_2+___8 c(Delete_C_N_2+___8) | DataE_N_2+___24 c(DataE_N_2+___24)
12. c(DataC_E_N_2+___10) → Edit_2+___6 c(Edit_2+___6) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Can-cel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | Delete_C_E_N_2+___15 c(Delete_C_E_N_2+___15) | DataE_E_N_2+___36 c(DataE_E_N_2+___36)
13. c(DataI_E_N_2+___11) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | Delete_I_E_N_2+___16 c(Delete_I_E_N_2+___16) | DataE_E_N_2+___36 c(DataE_E_N_2+___36)
14. c(Cancel_E_2+___12) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28)
15. c(Save_C_E_2+___13) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28)
16. c(Save_I_E_2+___14) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | De-lete_I_E_W_2+___38 c(Delete_I_E_W_2+___38)
17. c(Delete_C_E_N_2+___15) → DCancel_C_E_N_2+___17 c(DCancel_C_E_N_2+___17) | DOK_2+_1___20 c(DOK_2+_1___20)
18. c(Delete_I_E_N_2+___16) → DCancel_I_E_N_2+___18 c(DCancel_I_E_N_2+___18) | DOK_2+_1___20 c(DOK_2+_1___20)
19. c(DCancel_C_E_N_2+___17) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | Delete_C_E_N_2+___15 c(Delete_C_E_N_2+___15) | DataE_E_N_2+___36 c(DataE_E_N_2+___36)
20. c(DCancel_I_E_N_2+___18) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | Delete_I_E_N_2+___16 c(Delete_I_E_N_2+___16) | DataE_E_N_2+___36 c(DataE_E_N_2+___36)
21. c(]___19) → ε
22. c(DOK_2+_1___20) →]___19 c(]___19) | DataC_N_1___49 c(DataC_N_1___49) | DataI_N_1___50 c(DataI_N_1___50) | Edit_1___52 c(Edit_1___52) | Add_E_1___62 c(Add_E_1___62) | DataE_N_1___65 c(DataE_N_1___65) | Delete_E_N_1___69 c(Delete_E_N_1___69)

23. c(Add_E_2+___21) → Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_E_W_2+___29 c(Delete_E_W_2+___29)
24. c(DataC_W_2+___22) → Add_C_2+___1 c(Add_C_2+___1) | Edit_2+___6 c(Edit_2+___6) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_C_W_2+___27 c(Delete_C_W_2+___27)
25. c(DataI_W_2+___23) → Add_I_2+___2 c(Add_I_2+___2) | Edit_2+___6 c(Edit_2+___6) | DataC_W_2+___22 c(DataC_W_2+___22) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_I_W_2+___26 c(Delete_I_W_2+___26)
26. c(DataE_N_2+___24) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28)
27. c(DataE_W_2+___25) → Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_E_W_2+___29 c(Delete_E_W_2+___29)
28. c(Delete_I_W_2+___26) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_I_W_2+___30 c(DCancel_I_W_2+___30)
29. c(Delete_C_W_2+___27) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_C_W_2+___31 c(DCancel_C_W_2+___31)
30. c(Delete_E_N_2+___28) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_E_N_2+___32 c(DCancel_E_N_2+___32)
31. c(Delete_E_W_2+___29) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_E_W_2+___33 c(DCancel_E_W_2+___33)
32. c(DCancel_I_W_2+___30) → Add_I_2+___2 c(Add_I_2+___2) | Edit_2+___6 c(Edit_2+___6) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_I_W_2+___26 c(Delete_I_W_2+___26)
33. c(DCancel_C_W_2+___31) → Add_C_2+___1 c(Add_C_2+___1) | Edit_2+___6 c(Edit_2+___6) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_C_W_2+___27 c(Delete_C_W_2+___27)
34. c(DCancel_E_N_2+___32) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28)
35. c(DCancel_E_W_2+___33) → Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_E_W_2+___29 c(Delete_E_W_2+___29)
36. c(DataC_E_W_2+___34) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | Delete_C_E_W_2+___39 c(Delete_C_E_W_2+___39)
37. c(DataI_E_W_2+___35) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | Delete_I_E_W_2+___38 c(Delete_I_E_W_2+___38)

38. c(DataE_E_N_2+___36) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | DataE_E_N_2+___36 c(DataE_E_N_2+___36) | Delete_E_E_N_2+___40 c(Delete_E_E_N_2+___40) | Save_E_E_2+___46 c(Save_E_E_2+___46)
39. c(DataE_E_W_2+___37) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | Delete_E_E_W_2+___41 c(Delete_E_E_W_2+___41) | Save_E_E_2+___46 c(Save_E_E_2+___46)
40. c(Delete_I_E_W_2+___38) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_I_E_W_2+___42 c(DCancel_I_E_W_2+___42)
41. c(Delete_C_E_W_2+___39) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_C_E_W_2+___43 c(DCancel_C_E_W_2+___43)
42. c(Delete_E_E_N_2+___40) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_E_E_N_2+___44 c(DCancel_E_E_N_2+___44)
43. c(Delete_E_E_W_2+___41) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_E_E_W_2+___45 c(DCancel_E_E_W_2+___45)
44. c(DCancel_I_E_W_2+___42) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | De-lete_I_E_W_2+___38 c(Delete_I_E_W_2+___38)
45. c(DCancel_C_E_W_2+___43) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | De-lete_C_E_W_2+___39 c(Delete_C_E_W_2+___39)
46. c(DCancel_E_E_N_2+___44) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | DataE_E_N_2+___36 c(DataE_E_N_2+___36) | Delete_E_E_N_2+___40 c(Delete_E_E_N_2+___40) | Save_E_E_2+___46 c(Save_E_E_2+___46)
47. c(DCancel_E_E_W_2+___45) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | Delete_E_E_W_2+___41 c(Delete_E_E_W_2+___41) | Save_E_E_2+___46 c(Save_E_E_2+___46)
48. c(Save_E_E_2+___46) → Edit_2+___6 c(Edit_2+___6) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | De-lete_E_E_W_2+___41 c(Delete_E_E_W_2+___41) | Save_E_E_2+___46 c(Save_E_E_2+___46)
49. c(Add_C_1___47) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) |]___19 c(]___19) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28)

50. c(Add_I_1___48) → Add_I_1___48 c(Add_I_1___48) | Edit_1___52 c(Edit_1___52) | DataC_W_1___63 c(DataC_W_1___63) | DataI_W_1___64 c(DataI_W_1___64) | DataE_W_1___66 c(DataE_W_1___66) | De-lete_I_W_1___67 c(Delete_I_W_1___67)
51. c(DataC_N_1___49) → Add_C_1___47 c(Add_C_1___47) | DataI_N_1___50 c(DataI_N_1___50) | Edit_1___52 c(Edit_1___52) | Delete_C_N_1___54 c(Delete_C_N_1___54) | DataE_N_1___65 c(DataE_N_1___65)
52. c(DataI_N_1___50) → Add_I_1___48 c(Add_I_1___48) | DataC_N_1___49 c(DataC_N_1___49) | De-lete_I_N_1___51 c(Delete_I_N_1___51) | Edit_1___52 c(Edit_1___52) | DataE_N_1___65 c(DataE_N_1___65)
53. c(Delete_I_N_1___51) → DCancel_I_N_1___53 c(DCancel_I_N_1___53) | DOK_1_0___61 c(DOK_1_0___61)
54. c(Edit_1___52) → DataC_E_N_1___56 c(DataC_E_N_1___56) | DataI_E_N_1___57 c(DataI_E_N_1___57) | Cancel_E_1___58 c(Cancel_E_1___58) | Save_C_E_1___59 c(Save_C_E_1___59) | DataE_E_N_1___77 c(DataE_E_N_1___77)
55. c(DCancel_I_N_1___53) → Add_I_1___48 c(Add_I_1___48) | DataC_N_1___49 c(DataC_N_1___49) | DataI_N_1___50 c(DataI_N_1___50) | Delete_I_N_1___51 c(Delete_I_N_1___51) | Edit_1___52 c(Edit_1___52) | DataE_N_1___65 c(DataE_N_1___65)
56. c(Delete_C_N_1___54) → DCancel_C_N_1___55 c(DCancel_C_N_1___55) | DOK_1_0___61 c(DOK_1_0___61)
57. c(DCancel_C_N_1___55) → Add_C_1___47 c(Add_C_1___47) | DataC_N_1___49 c(DataC_N_1___49) | DataI_N_1___50 c(DataI_N_1___50) | Edit_1___52 c(Edit_1___52) | Delete_C_N_1___54 c(Delete_C_N_1___54) | DataE_N_1___65 c(DataE_N_1___65)
58. c(DataC_E_N_1___56) → DataI_E_N_1___57 c(DataI_E_N_1___57) | Cancel_E_1___58 c(Cancel_E_1___58) | Save_C_E_1___59 c(Save_C_E_1___59) | DataE_E_N_1___77 c(DataE_E_N_1___77)
59. c(DataI_E_N_1___57) → DataC_E_N_1___56 c(DataC_E_N_1___56) | Cancel_E_1___58 c(Cancel_E_1___58) | Save_I_E_1___60 c(Save_I_E_1___60) | DataE_E_N_1___77 c(DataE_E_N_1___77)
60. c(Cancel_E_1___58) → DataC_N_1___49 c(DataC_N_1___49) | DataI_N_1___50 c(DataI_N_1___50) | Edit_1___52 c(Edit_1___52) | Add_E_1___62 c(Add_E_1___62) | DataE_N_1___65 c(DataE_N_1___65) | De-lete_E_N_1___69 c(Delete_E_N_1___69)
61. c(Save_C_E_1___59) → DataC_N_1___49 c(DataC_N_1___49) | DataI_N_1___50 c(DataI_N_1___50) | Edit_1___52 c(Edit_1___52) | Add_E_1___62 c(Add_E_1___62) | DataE_N_1___65 c(DataE_N_1___65) | De-lete_E_N_1___69 c(Delete_E_N_1___69)
62. c(Save_I_E_1___60) → Cancel_E_1___58 c(Cancel_E_1___58) | Save_I_E_1___60 c(Save_I_E_1___60) | DataC_E_W_1___75 c(DataC_E_W_1___75) | DataI_E_W_1___76 c(DataI_E_W_1___76) | DataE_E_W_1___78 c(DataE_E_W_1___78)
63. c(DOK_1_0___61) →]___19 c(]___19) | DataC_N_0___82 c(DataC_N_0___82) | DataI_N_0___83 c(DataI_N_0___83) | Add_E_0___84 c(Add_E_0___84) | DataE_N_0___87 c(DataE_N_0___87)
64. c(Add_E_1___62) → Edit_1___52 c(Edit_1___52) | Add_E_1___62 c(Add_E_1___62) | DataC_W_1___63 c(DataC_W_1___63) | DataI_W_1___64 c(DataI_W_1___64) | DataE_W_1___66 c(DataE_W_1___66) | De-lete_E_W_1___70 c(Delete_E_W_1___70)
65. c(DataC_W_1___63) → Add_C_1___47 c(Add_C_1___47) | Edit_1___52 c(Edit_1___52) | DataI_W_1___64 c(DataI_W_1___64) | DataE_W_1___66 c(DataE_W_1___66) | Delete_C_W_1___68 c(Delete_C_W_1___68)

66. c(DataI_W_1___64) → Add_I_1___48 c(Add_I_1___48) | Edit_1___52 c(Edit_1___52) | DataC_W_1___63 c(DataC_W_1___63) | DataE_W_1___66 c(DataE_W_1___66) | Delete_I_W_1___67 c(Delete_I_W_1___67)
67. c(DataE_N_1___65) → DataC_N_1___49 c(DataC_N_1___49) | DataI_N_1___50 c(DataI_N_1___50) | Edit_1___52 c(Edit_1___52) | Add_E_1___62 c(Add_E_1___62) | DataE_N_1___65 c(DataE_N_1___65) | De-lete_E_N_1___69 c(Delete_E_N_1___69)
68. c(DataE_W_1___66) → Edit_1___52 c(Edit_1___52) | Add_E_1___62 c(Add_E_1___62) | DataC_W_1___63 c(DataC_W_1___63) | DataI_W_1___64 c(DataI_W_1___64) | DataE_W_1___66 c(DataE_W_1___66) | De-lete_E_W_1___70 c(Delete_E_W_1___70)
69. c(Delete_I_W_1___67) → DOK_1_0___61 c(DOK_1_0___61) | DCancel_I_W_1___71 c(DCancel_I_W_1___71)
70. c(Delete_C_W_1___68) → DOK_1_0___61 c(DOK_1_0___61) | DCancel_C_W_1___72 c(DCancel_C_W_1___72)
71. c(Delete_E_N_1___69) → DOK_1_0___61 c(DOK_1_0___61) | DCancel_E_N_1___73 c(DCancel_E_N_1___73)
72. c(Delete_E_W_1___70) → DOK_1_0___61 c(DOK_1_0___61) | DCancel_E_W_1___74 c(DCancel_E_W_1___74)
73. c(DCancel_I_W_1___71) → Add_I_1___48 c(Add_I_1___48) | Edit_1___52 c(Edit_1___52) | DataC_W_1___63 c(DataC_W_1___63) | DataI_W_1___64 c(DataI_W_1___64) | DataE_W_1___66 c(DataE_W_1___66) | Delete_I_W_1___67 c(Delete_I_W_1___67)
74. c(DCancel_C_W_1___72) → Add_C_1___47 c(Add_C_1___47) | Edit_1___52 c(Edit_1___52) | DataC_W_1___63 c(DataC_W_1___63) | DataI_W_1___64 c(DataI_W_1___64) | DataE_W_1___66 c(DataE_W_1___66) | Delete_C_W_1___68 c(Delete_C_W_1___68)
75. c(DCancel_E_N_1___73) → DataC_N_1___49 c(DataC_N_1___49) | DataI_N_1___50 c(DataI_N_1___50) | Edit_1___52 c(Edit_1___52) | Add_E_1___62 c(Add_E_1___62) | DataE_N_1___65 c(DataE_N_1___65) | De-lete_E_N_1___69 c(Delete_E_N_1___69)
76. c(DCancel_E_W_1___74) → Edit_1___52 c(Edit_1___52) | Add_E_1___62 c(Add_E_1___62) | DataC_W_1___63 c(DataC_W_1___63) | DataI_W_1___64 c(DataI_W_1___64) | DataE_W_1___66 c(DataE_W_1___66) | Delete_E_W_1___70 c(Delete_E_W_1___70)
77. c(DataC_E_W_1___75) → Cancel_E_1___58 c(Cancel_E_1___58) | Save_C_E_1___59 c(Save_C_E_1___59) | DataI_E_W_1___76 c(DataI_E_W_1___76) | DataE_E_W_1___78 c(DataE_E_W_1___78)
78. c(DataI_E_W_1___76) → Cancel_E_1___58 c(Cancel_E_1___58) | Save_I_E_1___60 c(Save_I_E_1___60) | DataC_E_W_1___75 c(DataC_E_W_1___75) | DataE_E_W_1___78 c(DataE_E_W_1___78)
79. c(DataE_E_N_1___77) → DataC_E_N_1___56 c(DataC_E_N_1___56) | DataI_E_N_1___57 c(DataI_E_N_1___57) | Cancel_E_1___58 c(Cancel_E_1___58) | DataE_E_N_1___77 c(DataE_E_N_1___77) | Save_E_E_1___79 c(Save_E_E_1___79)
80. c(DataE_E_W_1___78) → Cancel_E_1___58 c(Cancel_E_1___58) | DataC_E_W_1___75 c(DataC_E_W_1___75) | DataI_E_W_1___76 c(DataI_E_W_1___76) | DataE_E_W_1___78 c(DataE_E_W_1___78) | Save_E_E_1___79 c(Save_E_E_1___79)
81. c(Save_E_E_1___79) → DataC_E_W_1___75 c(DataC_E_W_1___75) | DataI_E_W_1___76 c(DataI_E_W_1___76) | DataE_E_W_1___78 c(DataE_E_W_1___78) | Save_E_E_1___79 c(Save_E_E_1___79)

82. c(Add_C_0___80) →]___19 c(]___19) | DataC_N_1___49 c(DataC_N_1___49) | DataI_N_1___50 c(DataI_N_1___50) | Edit_1___52 c(Edit_1___52) | Add_E_1___62 c(Add_E_1___62) | DataE_N_1___65 c(DataE_N_1___65) | Delete_E_N_1___69 c(Delete_E_N_1___69)
83. c(Add_I_0___81) → Add_I_0___81 c(Add_I_0___81) | DataC_W_0___85 c(DataC_W_0___85) | DataI_W_0___86 c(DataI_W_0___86) | DataE_W_0___88 c(DataE_W_0___88)
84. c(DataC_N_0___82) → Add_C_0___80 c(Add_C_0___80) | DataI_N_0___83 c(DataI_N_0___83) | DataE_N_0___87 c(DataE_N_0___87)
85. c(DataI_N_0___83) → Add_I_0___81 c(Add_I_0___81) | DataC_N_0___82 c(DataC_N_0___82) | DataE_N_0___87 c(DataE_N_0___87)
86. c(Add_E_0___84) → Add_E_0___84 c(Add_E_0___84) | DataC_W_0___85 c(DataC_W_0___85) | DataI_W_0___86 c(DataI_W_0___86) | DataE_W_0___88 c(DataE_W_0___88)
87. c(DataC_W_0___85) → Add_C_0___80 c(Add_C_0___80) | DataI_W_0___86 c(DataI_W_0___86) | DataE_W_0___88 c(DataE_W_0___88)
88. c(DataI_W_0___86) → Add_I_0___81 c(Add_I_0___81) | DataC_W_0___85 c(DataC_W_0___85) | DataE_W_0___88 c(DataE_W_0___88)
89. c(DataE_N_0___87) → DataC_N_0___82 c(DataC_N_0___82) | DataI_N_0___83 c(DataI_N_0___83) | Add_E_0___84 c(Add_E_0___84) | DataE_N_0___87 c(DataE_N_0___87)
90. c(DataE_W_0___88) → Add_E_0___84 c(Add_E_0___84) | DataC_W_0___85 c(DataC_W_0___85) | DataI_W_0___86 c(DataI_W_0___86) | DataE_W_0___88 c(DataE_W_0___88)
91. c(OpenSpecials_0___89) →]___19 c(]___19) | DataC_N_0___82 c(DataC_N_0___82) | DataI_N_0___83 c(DataI_N_0___83) | Add_E_0___84 c(Add_E_0___84) | DataE_N_0___87 c(DataE_N_0___87)

A.3 Productions of Additionals 1-Reg Model

1. S → [___0 c([___0)
2. c([___0) → OpenAdditionals_0___92 c(OpenAdditionals_0___92)
3. c(Add_C_2+___1) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) |]___19 c(]___19) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28) | AddEdit_2+___47 c(AddEdit_2+___47)
4. c(Add_I_2+___2) → Add_I_2+___2 c(Add_I_2+___2) | Edit_2+___6 c(Edit_2+___6) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_I_W_2+___26 c(Delete_I_W_2+___26) | AddEdit_2+___47 c(AddEdit_2+___47)
5. c(DataC_N_2+___3) → Add_C_2+___1 c(Add_C_2+___1) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Delete_C_N_2+___8 c(Delete_C_N_2+___8) | DataE_N_2+___24 c(DataE_N_2+___24) | AddEdit_2+___47 c(AddEdit_2+___47)

6. c(DataI_N_2+___4) → Add_I_2+___2 c(Add_I_2+___2) | DataC_N_2+___3 c(DataC_N_2+___3) | De-lete_I_N_2+___5 c(Delete_I_N_2+___5) | Edit_2+___6 c(Edit_2+___6) | DataE_N_2+___24 c(DataE_N_2+___24) | AddEdit_2+___47 c(AddEdit_2+___47)
7. c(Delete_I_N_2+___5) → DCancel_I_N_2+___7 c(DCancel_I_N_2+___7) | DOK_2+_1___20 c(DOK_2+_1___20)
8. c(Edit_2+___6) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | Delete_C_E_N_2+___15 c(Delete_C_E_N_2+___15) | DataE_E_N_2+___36 c(DataE_E_N_2+___36)
9. c(DCancel_I_N_2+___7) → Add_I_2+___2 c(Add_I_2+___2) | DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Delete_I_N_2+___5 c(Delete_I_N_2+___5) | Edit_2+___6 c(Edit_2+___6) | DataE_N_2+___24 c(DataE_N_2+___24) | AddEdit_2+___47 c(AddEdit_2+___47)
10. c(Delete_C_N_2+___8) → DCancel_C_N_2+___9 c(DCancel_C_N_2+___9) | DOK_2+_1___20 c(DOK_2+_1___20)
11. c(DCancel_C_N_2+___9) → Add_C_2+___1 c(Add_C_2+___1) | DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Delete_C_N_2+___8 c(Delete_C_N_2+___8) | DataE_N_2+___24 c(DataE_N_2+___24) | AddEdit_2+___47 c(AddEdit_2+___47)
12. c(DataC_E_N_2+___10) → Edit_2+___6 c(Edit_2+___6) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Can-cel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | Delete_C_E_N_2+___15 c(Delete_C_E_N_2+___15) | DataE_E_N_2+___36 c(DataE_E_N_2+___36) | AddEdit_2+___47 c(AddEdit_2+___47)
13. c(DataI_E_N_2+___11) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | Delete_I_E_N_2+___16 c(Delete_I_E_N_2+___16) | DataE_E_N_2+___36 c(DataE_E_N_2+___36) | AddEdit_2+___47 c(AddEdit_2+___47)
14. c(Cancel_E_2+___12) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28) | AddEdit_2+___47 c(AddEdit_2+___47)
15. c(Save_C_E_2+___13) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28) | AddEdit_2+___47 c(AddEdit_2+___47)
16. c(Save_I_E_2+___14) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | De-lete_I_E_W_2+___38 c(Delete_I_E_W_2+___38) | AddEdit_2+___47 c(AddEdit_2+___47)
17. c(Delete_C_E_N_2+___15) → DCancel_C_E_N_2+___17 c(DCancel_C_E_N_2+___17) | DOK_2+_1___20 c(DOK_2+_1___20)
18. c(Delete_I_E_N_2+___16) → DCancel_I_E_N_2+___18 c(DCancel_I_E_N_2+___18) | DOK_2+_1___20 c(DOK_2+_1___20)

19. c(DCancel_C_E_N_2+___17) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | Delete_C_E_N_2+___15 c(Delete_C_E_N_2+___15) | DataE_E_N_2+___36 c(DataE_E_N_2+___36) | AddEdit_2+___47 c(AddEdit_2+___47)
20. c(DCancel_I_E_N_2+___18) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | Delete_I_E_N_2+___16 c(Delete_I_E_N_2+___16) | DataE_E_N_2+___36 c(DataE_E_N_2+___36) | AddEdit_2+___47 c(AddEdit_2+___47)
21. c(]___19) → ε
22. c(DOK_2+_1___20) →]___19 c(]___19) | DataC_N_1___50 c(DataC_N_1___50) | DataI_N_1___51 c(DataI_N_1___51) | Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataE_N_1___66 c(DataE_N_1___66) | Delete_E_N_1___70 c(Delete_E_N_1___70) | AddEdit_1___81 c(AddEdit_1___81)
23. c(Add_E_2+___21) → Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_E_W_2+___29 c(Delete_E_W_2+___29) | AddEdit_2+___47 c(AddEdit_2+___47)
24. c(DataC_W_2+___22) → Add_C_2+___1 c(Add_C_2+___1) | Edit_2+___6 c(Edit_2+___6) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_C_W_2+___27 c(Delete_C_W_2+___27) | AddEdit_2+___47 c(AddEdit_2+___47)
25. c(DataI_W_2+___23) → Add_I_2+___2 c(Add_I_2+___2) | Edit_2+___6 c(Edit_2+___6) | DataC_W_2+___22 c(DataC_W_2+___22) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_I_W_2+___26 c(Delete_I_W_2+___26) | AddEdit_2+___47 c(AddEdit_2+___47)
26. c(DataE_N_2+___24) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28) | AddEdit_2+___47 c(AddEdit_2+___47)
27. c(DataE_W_2+___25) → Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_E_W_2+___29 c(Delete_E_W_2+___29) | AddEdit_2+___47 c(AddEdit_2+___47)
28. c(Delete_I_W_2+___26) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_I_W_2+___30 c(DCancel_I_W_2+___30)
29. c(Delete_C_W_2+___27) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_C_W_2+___31 c(DCancel_C_W_2+___31)
30. c(Delete_E_N_2+___28) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_E_N_2+___32 c(DCancel_E_N_2+___32)
31. c(Delete_E_W_2+___29) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_E_W_2+___33 c(DCancel_E_W_2+___33)
32. c(DCancel_I_W_2+___30) → Add_I_2+___2 c(Add_I_2+___2) | Edit_2+___6 c(Edit_2+___6) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_I_W_2+___26 c(Delete_I_W_2+___26) | AddEdit_2+___47 c(AddEdit_2+___47)
33. c(DCancel_C_W_2+___31) → Add_C_2+___1 c(Add_C_2+___1) | Edit_2+___6 c(Edit_2+___6) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23

c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_C_W_2+___27 c(Delete_C_W_2+___27) | AddEdit_2+___47 c(AddEdit_2+___47)

34. c(DCancel_E_N_2+___32) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28) | AddEdit_2+___47 c(AddEdit_2+___47)
35. c(DCancel_E_W_2+___33) → Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataC_W_2+___22 c(DataC_W_2+___22) | DataI_W_2+___23 c(DataI_W_2+___23) | DataE_W_2+___25 c(DataE_W_2+___25) | Delete_E_W_2+___29 c(Delete_E_W_2+___29) | AddEdit_2+___47 c(AddEdit_2+___47)
36. c(DataC_E_W_2+___34) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | Delete_C_E_W_2+___39 c(Delete_C_E_W_2+___39) | Add-Edit_2+___47 c(AddEdit_2+___47)
37. c(DataI_E_W_2+___35) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | Delete_I_E_W_2+___38 c(Delete_I_E_W_2+___38) | Add-Edit_2+___47 c(AddEdit_2+___47)
38. c(DataE_E_N_2+___36) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | DataE_E_N_2+___36 c(DataE_E_N_2+___36) | Delete_E_E_N_2+___40 c(Delete_E_E_N_2+___40) | Save_E_E_2+___46 c(Save_E_E_2+___46) | AddEdit_2+___47 c(AddEdit_2+___47)
39. c(DataE_E_W_2+___37) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | Delete_E_E_W_2+___41 c(Delete_E_E_W_2+___41) | Save_E_E_2+___46 c(Save_E_E_2+___46) | AddEdit_2+___47 c(AddEdit_2+___47)
40. c(Delete_I_E_W_2+___38) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_I_E_W_2+___42 c(DCancel_I_E_W_2+___42)
41. c(Delete_C_E_W_2+___39) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_C_E_W_2+___43 c(DCancel_C_E_W_2+___43)
42. c(Delete_E_E_N_2+___40) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_E_E_N_2+___44 c(DCancel_E_E_N_2+___44)
43. c(Delete_E_E_W_2+___41) → DOK_2+_1___20 c(DOK_2+_1___20) | DCancel_E_E_W_2+___45 c(DCancel_E_E_W_2+___45)
44. c(DCancel_I_E_W_2+___42) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_I_E_2+___14 c(Save_I_E_2+___14) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | De-lete_I_E_W_2+___38 c(Delete_I_E_W_2+___38) | AddEdit_2+___47 c(AddEdit_2+___47)
45. c(DCancel_C_E_W_2+___43) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | Save_C_E_2+___13 c(Save_C_E_2+___13) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) |

DataE_E_W_2+___37 c(DataE_E_W_2+___37) | De-lete_C_E_W_2+___39 c(Delete_C_E_W_2+___39) | AddEdit_2+___47 c(AddEdit_2+___47)

46. c(DCancel_E_E_N_2+___44) → Edit_2+___6 c(Edit_2+___6) | DataC_E_N_2+___10 c(DataC_E_N_2+___10) | DataI_E_N_2+___11 c(DataI_E_N_2+___11) | Cancel_E_2+___12 c(Cancel_E_2+___12) | DataE_E_N_2+___36 c(DataE_E_N_2+___36) | Delete_E_E_N_2+___40 c(Delete_E_E_N_2+___40) | Save_E_E_2+___46 c(Save_E_E_2+___46) | AddEdit_2+___47 c(AddEdit_2+___47)
47. c(DCancel_E_E_W_2+___45) → Edit_2+___6 c(Edit_2+___6) | Cancel_E_2+___12 c(Cancel_E_2+___12) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | Delete_E_E_W_2+___41 c(Delete_E_E_W_2+___41) | Save_E_E_2+___46 c(Save_E_E_2+___46) | AddEdit_2+___47 c(AddEdit_2+___47)
48. c(Save_E_E_2+___46) → Edit_2+___6 c(Edit_2+___6) | DataC_E_W_2+___34 c(DataC_E_W_2+___34) | DataI_E_W_2+___35 c(DataI_E_W_2+___35) | DataE_E_W_2+___37 c(DataE_E_W_2+___37) | De-lete_E_E_W_2+___41 c(Delete_E_E_W_2+___41) | Save_E_E_2+___46 c(Save_E_E_2+___46) | Add-Edit_2+___47 c(AddEdit_2+___47)
49. c(AddEdit_2+___47) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28)
50. c(Add_C_1___48) → DataC_N_2+___3 c(DataC_N_2+___3) | DataI_N_2+___4 c(DataI_N_2+___4) | Edit_2+___6 c(Edit_2+___6) |]___19 c(]___19) | Add_E_2+___21 c(Add_E_2+___21) | DataE_N_2+___24 c(DataE_N_2+___24) | Delete_E_N_2+___28 c(Delete_E_N_2+___28) | AddEdit_2+___47 c(AddEdit_2+___47)
51. c(Add_I_1___49) → Add_I_1___49 c(Add_I_1___49) | Edit_1___53 c(Edit_1___53) | DataC_W_1___64 c(DataC_W_1___64) | DataI_W_1___65 c(DataI_W_1___65) | DataE_W_1___67 c(DataE_W_1___67) | De-lete_I_W_1___68 c(Delete_I_W_1___68) | AddEdit_1___81 c(AddEdit_1___81)
52. c(DataC_N_1___50) → Add_C_1___48 c(Add_C_1___48) | DataI_N_1___51 c(DataI_N_1___51) | Edit_1___53 c(Edit_1___53) | Delete_C_N_1___55 c(Delete_C_N_1___55) | DataE_N_1___66 c(DataE_N_1___66) | AddEdit_1___81 c(AddEdit_1___81)
53. c(DataI_N_1___51) → Add_I_1___49 c(Add_I_1___49) | DataC_N_1___50 c(DataC_N_1___50) | De-lete_I_N_1___52 c(Delete_I_N_1___52) | Edit_1___53 c(Edit_1___53) | DataE_N_1___66 c(DataE_N_1___66) | AddEdit_1___81 c(AddEdit_1___81)
54. c(Delete_I_N_1___52) → DCancel_I_N_1___54 c(DCancel_I_N_1___54) | DOK_1_0___62 c(DOK_1_0___62)
55. c(Edit_1___53) → DataC_E_N_1___57 c(DataC_E_N_1___57) | DataI_E_N_1___58 c(DataI_E_N_1___58) | Cancel_E_1___59 c(Cancel_E_1___59) | Save_C_E_1___60 c(Save_C_E_1___60) | DataE_E_N_1___78 c(DataE_E_N_1___78)
56. c(DCancel_I_N_1___54) → Add_I_1___49 c(Add_I_1___49) | DataC_N_1___50 c(DataC_N_1___50) | DataI_N_1___51 c(DataI_N_1___51) | Delete_I_N_1___52 c(Delete_I_N_1___52) | Edit_1___53 c(Edit_1___53) | DataE_N_1___66 c(DataE_N_1___66) | AddEdit_1___81 c(AddEdit_1___81)
57. c(Delete_C_N_1___55) → DCancel_C_N_1___56 c(DCancel_C_N_1___56) | DOK_1_0___62 c(DOK_1_0___62)

58. c(DCancel_C_N_1___56) → Add_C_1___48 c(Add_C_1___48) | DataC_N_1___50 c(DataC_N_1___50) | DataI_N_1___51 c(DataI_N_1___51) | Edit_1___53 c(Edit_1___53) | Delete_C_N_1___55 c(Delete_C_N_1___55) | DataE_N_1___66 c(DataE_N_1___66) | AddEdit_1___81 c(AddEdit_1___81)
59. c(DataC_E_N_1___57) → DataI_E_N_1___58 c(DataI_E_N_1___58) | Cancel_E_1___59 c(Cancel_E_1___59) | Save_C_E_1___60 c(Save_C_E_1___60) | DataE_E_N_1___78 c(DataE_E_N_1___78) | AddEdit_1___81 c(AddEdit_1___81)
60. c(DataI_E_N_1___58) → DataC_E_N_1___57 c(DataC_E_N_1___57) | Cancel_E_1___59 c(Cancel_E_1___59) | Save_I_E_1___61 c(Save_I_E_1___61) | DataE_E_N_1___78 c(DataE_E_N_1___78) | AddEdit_1___81 c(AddEdit_1___81)
61. c(Cancel_E_1___59) → DataC_N_1___50 c(DataC_N_1___50) | DataI_N_1___51 c(DataI_N_1___51) | Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataE_N_1___66 c(DataE_N_1___66) | De-lete_E_N_1___70 c(Delete_E_N_1___70) | AddEdit_1___81 c(AddEdit_1___81)
62. c(Save_C_E_1___60) → DataC_N_1___50 c(DataC_N_1___50) | DataI_N_1___51 c(DataI_N_1___51) | Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataE_N_1___66 c(DataE_N_1___66) | De-lete_E_N_1___70 c(Delete_E_N_1___70) | AddEdit_1___81 c(AddEdit_1___81)
63. c(Save_I_E_1___61) → Cancel_E_1___59 c(Cancel_E_1___59) | Save_I_E_1___61 c(Save_I_E_1___61) | DataC_E_W_1___76 c(DataC_E_W_1___76) | DataI_E_W_1___77 c(DataI_E_W_1___77) | DataE_E_W_1___79 c(DataE_E_W_1___79) | AddEdit_1___81 c(AddEdit_1___81)
64. c(DOK_1_0___62) →]___19 c(]___19) | DataC_N_0___84 c(DataC_N_0___84) | DataI_N_0___85 c(DataI_N_0___85) | Add_E_0___86 c(Add_E_0___86) | DataE_N_0___89 c(DataE_N_0___89) | Add-Edit_0___91 c(AddEdit_0___91)
65. c(Add_E_1___63) → Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataC_W_1___64 c(DataC_W_1___64) | DataI_W_1___65 c(DataI_W_1___65) | DataE_W_1___67 c(DataE_W_1___67) | De-lete_E_W_1___71 c(Delete_E_W_1___71) | AddEdit_1___81 c(AddEdit_1___81)
66. c(DataC_W_1___64) → Add_C_1___48 c(Add_C_1___48) | Edit_1___53 c(Edit_1___53) | DataI_W_1___65 c(DataI_W_1___65) | DataE_W_1___67 c(DataE_W_1___67) | Delete_C_W_1___69 c(Delete_C_W_1___69) | AddEdit_1___81 c(AddEdit_1___81)
67. c(DataI_W_1___65) → Add_I_1___49 c(Add_I_1___49) | Edit_1___53 c(Edit_1___53) | DataC_W_1___64 c(DataC_W_1___64) | DataE_W_1___67 c(DataE_W_1___67) | Delete_I_W_1___68 c(Delete_I_W_1___68) | AddEdit_1___81 c(AddEdit_1___81)
68. c(DataE_N_1___66) → DataC_N_1___50 c(DataC_N_1___50) | DataI_N_1___51 c(DataI_N_1___51) | Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataE_N_1___66 c(DataE_N_1___66) | De-lete_E_N_1___70 c(Delete_E_N_1___70) | AddEdit_1___81 c(AddEdit_1___81)
69. c(DataE_W_1___67) → Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataC_W_1___64 c(DataC_W_1___64) | DataI_W_1___65 c(DataI_W_1___65) | DataE_W_1___67 c(DataE_W_1___67) | De-lete_E_W_1___71 c(Delete_E_W_1___71) | AddEdit_1___81 c(AddEdit_1___81)
70. c(Delete_I_W_1___68) → DOK_1_0___62 c(DOK_1_0___62) | DCancel_I_W_1___72 c(DCancel_I_W_1___72)
71. c(Delete_C_W_1___69) → DOK_1_0___62 c(DOK_1_0___62) | DCancel_C_W_1___73 c(DCancel_C_W_1___73)

72. c(Delete_E_N_1___70) → DOK_1_0___62 c(DOK_1_0___62) | DCancel_E_N_1___74 c(DCancel_E_N_1___74)
73. c(Delete_E_W_1___71) → DOK_1_0___62 c(DOK_1_0___62) | DCancel_E_W_1___75 c(DCancel_E_W_1___75)
74. c(DCancel_I_W_1___72) → Add_I_1___49 c(Add_I_1___49) | Edit_1___53 c(Edit_1___53) | DataC_W_1___64 c(DataC_W_1___64) | DataI_W_1___65 c(DataI_W_1___65) | DataE_W_1___67 c(DataE_W_1___67) | Delete_I_W_1___68 c(Delete_I_W_1___68) | AddEdit_1___81 c(AddEdit_1___81)
75. c(DCancel_C_W_1___73) → Add_C_1___48 c(Add_C_1___48) | Edit_1___53 c(Edit_1___53) | DataC_W_1___64 c(DataC_W_1___64) | DataI_W_1___65 c(DataI_W_1___65) | DataE_W_1___67 c(DataE_W_1___67) | Delete_C_W_1___69 c(Delete_C_W_1___69) | AddEdit_1___81 c(AddEdit_1___81)
76. c(DCancel_E_N_1___74) → DataC_N_1___50 c(DataC_N_1___50) | DataI_N_1___51 c(DataI_N_1___51) | Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataE_N_1___66 c(DataE_N_1___66) | De-lete_E_N_1___70 c(Delete_E_N_1___70) | AddEdit_1___81 c(AddEdit_1___81)
77. c(DCancel_E_W_1___75) → Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataC_W_1___64 c(DataC_W_1___64) | DataI_W_1___65 c(DataI_W_1___65) | DataE_W_1___67 c(DataE_W_1___67) | Delete_E_W_1___71 c(Delete_E_W_1___71) | AddEdit_1___81 c(AddEdit_1___81)
78. c(DataC_E_W_1___76) → Cancel_E_1___59 c(Cancel_E_1___59) | Save_C_E_1___60 c(Save_C_E_1___60) | DataI_E_W_1___77 c(DataI_E_W_1___77) | DataE_E_W_1___79 c(DataE_E_W_1___79) | AddEdit_1___81 c(AddEdit_1___81)
79. c(DataI_E_W_1___77) → Cancel_E_1___59 c(Cancel_E_1___59) | Save_I_E_1___61 c(Save_I_E_1___61) | DataC_E_W_1___76 c(DataC_E_W_1___76) | DataE_E_W_1___79 c(DataE_E_W_1___79) | AddEdit_1___81 c(AddEdit_1___81)
80. c(DataE_E_N_1___78) → DataC_E_N_1___57 c(DataC_E_N_1___57) | DataI_E_N_1___58 c(DataI_E_N_1___58) | Cancel_E_1___59 c(Cancel_E_1___59) | DataE_E_N_1___78 c(DataE_E_N_1___78) | Save_E_E_1___80 c(Save_E_E_1___80) | AddEdit_1___81 c(AddEdit_1___81)
81. c(DataE_E_W_1___79) → Cancel_E_1___59 c(Cancel_E_1___59) | DataC_E_W_1___76 c(DataC_E_W_1___76) | DataI_E_W_1___77 c(DataI_E_W_1___77) | DataE_E_W_1___79 c(DataE_E_W_1___79) | Save_E_E_1___80 c(Save_E_E_1___80) | AddEdit_1___81 c(AddEdit_1___81)
82. c(Save_E_E_1___80) → DataC_E_W_1___76 c(DataC_E_W_1___76) | DataI_E_W_1___77 c(DataI_E_W_1___77) | DataE_E_W_1___79 c(DataE_E_W_1___79) | Save_E_E_1___80 c(Save_E_E_1___80) | AddEdit_1___81 c(AddEdit_1___81)
83. c(AddEdit_1___81) → DataC_N_1___50 c(DataC_N_1___50) | DataI_N_1___51 c(DataI_N_1___51) | Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataE_N_1___66 c(DataE_N_1___66) | De-lete_E_N_1___70 c(Delete_E_N_1___70)
84. c(Add_C_0___82) →]___19 c(]___19) | DataC_N_1___50 c(DataC_N_1___50) | DataI_N_1___51 c(DataI_N_1___51) | Edit_1___53 c(Edit_1___53) | Add_E_1___63 c(Add_E_1___63) | DataE_N_1___66 c(DataE_N_1___66) | Delete_E_N_1___70 c(Delete_E_N_1___70) | AddEdit_1___81 c(AddEdit_1___81)
85. c(Add_I_0___83) → Add_I_0___83 c(Add_I_0___83) | DataC_W_0___87 c(DataC_W_0___87) | DataI_W_0___88 c(DataI_W_0___88) | DataE_W_0___90 c(DataE_W_0___90) | AddEdit_0___91 c(AddEdit_0___91)

86. c(DataC_N_0___84) → Add_C_0___82 c(Add_C_0___82) | DataI_N_0___85 c(DataI_N_0___85) | DataE_N_0___89 c(DataE_N_0___89) | AddEdit_0___91 c(AddEdit_0___91)
87. c(DataI_N_0___85) → Add_I_0___83 c(Add_I_0___83) | DataC_N_0___84 c(DataC_N_0___84) | DataE_N_0___89 c(DataE_N_0___89) | AddEdit_0___91 c(AddEdit_0___91)
88. c(Add_E_0___86) → Add_E_0___86 c(Add_E_0___86) | DataC_W_0___87 c(DataC_W_0___87) | DataI_W_0___88 c(DataI_W_0___88) | DataE_W_0___90 c(DataE_W_0___90) | AddEdit_0___91 c(AddEdit_0___91)
89. c(DataC_W_0___87) → Add_C_0___82 c(Add_C_0___82) | DataI_W_0___88 c(DataI_W_0___88) | DataE_W_0___90 c(DataE_W_0___90) | AddEdit_0___91 c(AddEdit_0___91)
90. c(DataI_W_0___88) → Add_I_0___83 c(Add_I_0___83) | DataC_W_0___87 c(DataC_W_0___87) | DataE_W_0___90 c(DataE_W_0___90) | AddEdit_0___91 c(AddEdit_0___91)
91. c(DataE_N_0___89) → DataC_N_0___84 c(DataC_N_0___84) | DataI_N_0___85 c(DataI_N_0___85) | Add_E_0___86 c(Add_E_0___86) | DataE_N_0___89 c(DataE_N_0___89) | AddEdit_0___91 c(AddEdit_0___91)
92. c(DataE_W_0___90) → Add_E_0___86 c(Add_E_0___86) | DataC_W_0___87 c(DataC_W_0___87) | DataI_W_0___88 c(DataI_W_0___88) | DataE_W_0___90 c(DataE_W_0___90) | AddEdit_0___91 c(AddEdit_0___91)
93. c(AddEdit_0___91) → DataC_N_0___84 c(DataC_N_0___84) | DataI_N_0___85 c(DataI_N_0___85) | Add_E_0___86 c(Add_E_0___86) | DataE_N_0___89 c(DataE_N_0___89)
94. c(OpenAdditionals_0___92) →]___19 c(]___19) | DataC_N_0___84 c(DataC_N_0___84) | DataI_N_0___85 c(DataI_N_0___85) | Add_E_0___86 c(Add_E_0___86) | DataE_N_0___89 c(DataE_N_0___89) | Add-Edit_0___91 c(AddEdit_0___91)

B Data for Case Studies

Table B.1. Test Generation Data for ShearBar

ShearBar				
Test Set	**Sequence Number**	**Total Length**	**Average Length**	**Test Generation Time (s)**
1-Reg	32439	1125004	34.68	6
M-1-Reg	32465	1126621	34.70	6
Random(2)	32759	1306973	39.90	49756
ESG(2,1)	42948	1125580	26.21	89645
ESG(2,M-1)	43039	1126818	26.18	89958
MK(1)	30780	1924440	62.52	42927
MK(M-1)	30934	1932476	62.47	43174
2-Reg	40754	1465701	35.96	12
M-2-Reg	40764	1466319	37.22	12
Random(3)	41183	1697887	41.23	54756
ESG(3,2)	46292	1466650	31.68	80061
ESG(3,M-2)	46292	1466650	31.68	80189
MK(2)	40035	2514171	62.8	56819
MK(M-2)	40035	2514171	62.8	56706
3-Reg	52188	1942081	37.21	19
M-3-Reg	52232	1944064	35.97	20
Random(4)	52727	2231070	42.31	95606
ESG(4,3)	62400	1942553	31.13	88687
ESG(4,M-3)	62481	1944104	31.12	88159
MK(3)	52377	3323226	63.45	74683
MK(M-3)	52512	3330828	63.43	74856

Table B.2. Test Generation Data for Specials

Specials				
Test Set	**Sequence Number**	**Total Length**	**Average Length**	**Test Generation Time (s)**
1-Reg	832	7387	8.88	1
M-1-Reg	1349	16358	12.13	3
Random(2)	1182	25852	21.87	746
ESG(2,1)	372	7419	19.94	10
ESG(2,M-1)	871	16406	18.84	24
MK(1)	513	13094	25.52	12
MK(M-1)	995	25327	25.45	18
2-Reg	3972	41548	10.46	4
M-2-Reg	7584	100593	13.26	49
Random(3)	5563	127968	23.00	12211
ESG(3,2)	2187	42910	19.62	44
ESG(3,M-2)	5187	103678	19.99	148
MK(2)	2684	65775	24.51	44
MK(M-2)	6268	153337	24.46	99
3-Reg	21438	250383	11.68	58
M-3-Reg	39878	576675	14.46	3928
Random(4)	28404	677718	23.86	166003
ESG(4,3)	15313	253109	16.53	1322
ESG(4,M-3)	39506	582182	14.74	5022
MK(3)	16324	395166	24.21	272
MK(M-3)	38829	928670	23.92	722

Table B.3. Test Generation Data for Additionals

Additionals				
Test Set	**Sequence Number**	**Total Length**	**Average Length**	**Test Generation Time (s)**
1-Reg	910	8154	8.96	1
M-1-Reg	1684	21217	12.60	5
Random(2)	1315	29462	22.40	3967
ESG(2,1)	680	8786	12.92	13
ESG(2,M-1)	1749	21280	12.17	50
MK(1)	737	15776	21.41	9
MK(M-1)	1519	33982	22.37	23
2-Reg	5069	53172	10.49	6
M-2-Reg	11019	148652	13.49	150
Random(3)	7167	167999	23.44	71991
ESG(3,2)	3491	53984	15.46	67
ESG(3,M-2)	10022	149850	14.95	344
MK(2)	3803	83448	21.94	55
MK(M-2)	9587	219853	22.93	131
3-Reg	31284	364059	11.64	170
M-3-Reg	65644	959294	14.61	12930
Random(4)	41691	1010872	24.25	1034561
ESG(4,3)	28665	377967	13.19	3168
ESG(4,M-3)	89023	961452	10.80	19091
MK(3)	23654	542688	22.94	351
MK(M-3)	62794	1434617	22.85	1087

Table B.4. Test Execution Data for ShearBar

ShearBar			
Test Set	**Events Executed**	**Faults Revealed**	**Fault Detection Rate**
1-Reg	1131355	165	0.000145843
M-1-Reg	1133560	174	0.000153499
Random(2)	1313915	174.75	0.000133051
ESG(2,1)	1131290	134	0.000118449
ESG(2,M-1)	1132573	135	0.000119198
MK(1)	1129508	103	0.000091190
MK(M-1)	1132958	103	0.000090912
2-Reg	1473026	182	0.000123555
M-2-Reg	1473765	184	0.000124850
Random(3)	1705243.25	183	0.000107362
ESG(3,2)	1472199	131	0.000088983
ESG(3,M-2)	1472199	131	0.000088983
MK(2)	1472884	103	0.000069931
MK(M-2)	1472884	103	0.000069931
3-Reg	1949959	194	0.000099489
M-3-Reg	1951997	195	0.000099898
Random(4)	2238921	194	0.000086684
ESG(4,3)	1948306	138	0.000070831
ESG(4,M-3)	1949857	138	0.000070774
MK(3)	1947314	108	0.000055461
MK(M-3)	1951490	108	0.000055342

Table B.5. Test Execution Data for Specials

Specials			
Test Set	**Events Executed**	**Faults Revealed**	**Fault Detection Rate**
1-Reg	8613	72	0.008359457
M-1-Reg	18407	96	0.005215407
Random(2)	27729.5	95.25	0.003631357
ESG(2,1)	8501	46	0.005411128
ESG(2,M-1)	17652	54	0.003059143
MK(1)	12116	40	0.003301420
MK(M-1)	20127	43	0.002136434
2-Reg	43851	122	0.002782149
M-2-Reg	103638	147	0.001418399
Random(3)	130738	135.25	0.001093049
ESG(3,2)	45146	74	0.001639126
ESG(3,M-2)	106085	83	0.000782391
MK(2)	46237	52	0.001124640
MK(M-2)	106033	69	0.000650741
3-Reg	253668	166	0.000654399
M-3-Reg	580495	178	0.000306635
Random(4)	681379.75	172.5	0.000268535
ESG(4,3)	255789	97	0.000379219
ESG(4,M-3)	585078	107	0.000182882
MK(3)	256265	83	0.000323883
MK(M-3)	581852	98	0.000168428

Table B.6. Test Execution Data for Additionals

Additionals			
Test Set	**Events Executed**	**Faults Revealed**	**Fault Detection Rate**
1-Reg	9154	65	0.007100721
M-1-Reg	22909	92	0.004015889
Random(2)	31191.75	86.75	0.002950497
ESG(2,1)	9573	38	0.003969498
ESG(2,M-1)	22474	50	0.002224793
MK(1)	11008	38	0.003452035
MK(M-1)	22796	41	0.001798561
2-Reg	55086	114	0.002069491
M-2-Reg	151403	139	0.000918080
Random(3)	170767.5	129.25	0.000801930
ESG(3,2)	55452	64	0.001154151
ESG(3,M-2)	151544	78	0.000514702
MK(2)	57384	57	0.000993308
MK(M-2)	151339	67	0.000442715
3-Reg	367094	161	0.000438580
M-3-Reg	962656	177	0.000183866
Random(4)	1014664.5	172.25	0.000180284
ESG(4,3)	380353	93	0.000244510
ESG(4,M-3)	964173	110	0.000114087
MK(3)	369004	81	0.000219510
MK(M-3)	965881	100	0.000103532

Table B.7. Faults Revealed for ShearBar

ShearBar				
Test Set	**m=1**	**m=2**	**m=3**	**m=4**
1-Reg	50	47	35	33
M-1-Reg	50	47	39	38
Random(2)	50.00	47.25	39.70	37.50
ESG(2,1)	34	36	30	34
ESG(2,M-1)	34	37	30	34
MK(1)	28	25	26	24
MK(M-1)	28	25	26	24
2-Reg	50	50	44	38
M-2-Reg	50	50	45	39
Random(3)	50.00	50.00	43.42	39.42
ESG(3,2)	33	36	32	30
ESG(3,M-2)	33	36	32	30
MK(2)	28	25	26	24
MK(M-2)	28	25	26	24
3-Reg	50	50	50	44
M-3-Reg	50	50	50	45
Random(4)	50.00	50.00	50.00	44.08
ESG(4,3)	33	36	35	34
ESG(4,M-3)	33	36	35	34
MK(3)	28	27	28	25
MK(M-3)	28	27	28	25

Table B.8. Faults Revealed for Specials

Specials				
Test Set	**m=1**	**m=2**	**m=3**	**m=4**
1-Reg	50	16	6	0
M-1-Reg	50	32	12	2
Random(2)	50.00	29.62	11.29	4.20
ESG(2,1)	29	10	5	2
ESG(2,M-1)	31	14	7	2
MK(1)	25	10	2	3
MK(M-1)	26	11	2	4
2-Reg	50	50	18	4
M-2-Reg	50	50	33	14
Random(3)	50.00	50.00	23.46	11.82
ESG(3,2)	28	25	15	6
ESG(3,M-2)	34	26	17	6
MK(2)	29	15	4	4
MK(M-2)	34	22	8	5
3-Reg	49	49	50	18
M-3-Reg	49	49	50	30
Random(4)	49.00	49.00	50.00	24.45
ESG(4,3)	28	27	27	15
ESG(4,M-3)	35	30	27	15
MK(3)	36	28	12	7
MK(M-3)	40	34	16	8

Table B.9. Faults Revealed for Additionals

Additionals				
Test Set	**m=1**	**m=2**	**m=3**	**m=4**
1-Reg	50	13	2	0
M-1-Reg	50	32	8	2
Random(2)	50.00	24.43	8.79	3.52
ESG(2,1)	27	9	2	0
ESG(2,M-1)	32	15	3	0
MK(1)	25	11	2	0
MK(M-1)	26	12	3	0
2-Reg	49	50	12	3
M-2-Reg	49	50	30	10
Random(3)	49.00	50.00	18.89	10.96
ESG(3,2)	26	26	10	2
ESG(3,M-2)	30	32	13	3
MK(2)	31	20	3	3
MK(M-2)	33	25	4	5
3-Reg	48	50	50	13
M-3-Reg	48	50	50	29
Random(4)	48.00	50.00	50.00	24.09
ESG(4,3)	27	27	25	14
ESG(4,M-3)	32	33	27	18
MK(3)	35	31	8	7
MK(M-3)	38	36	15	11

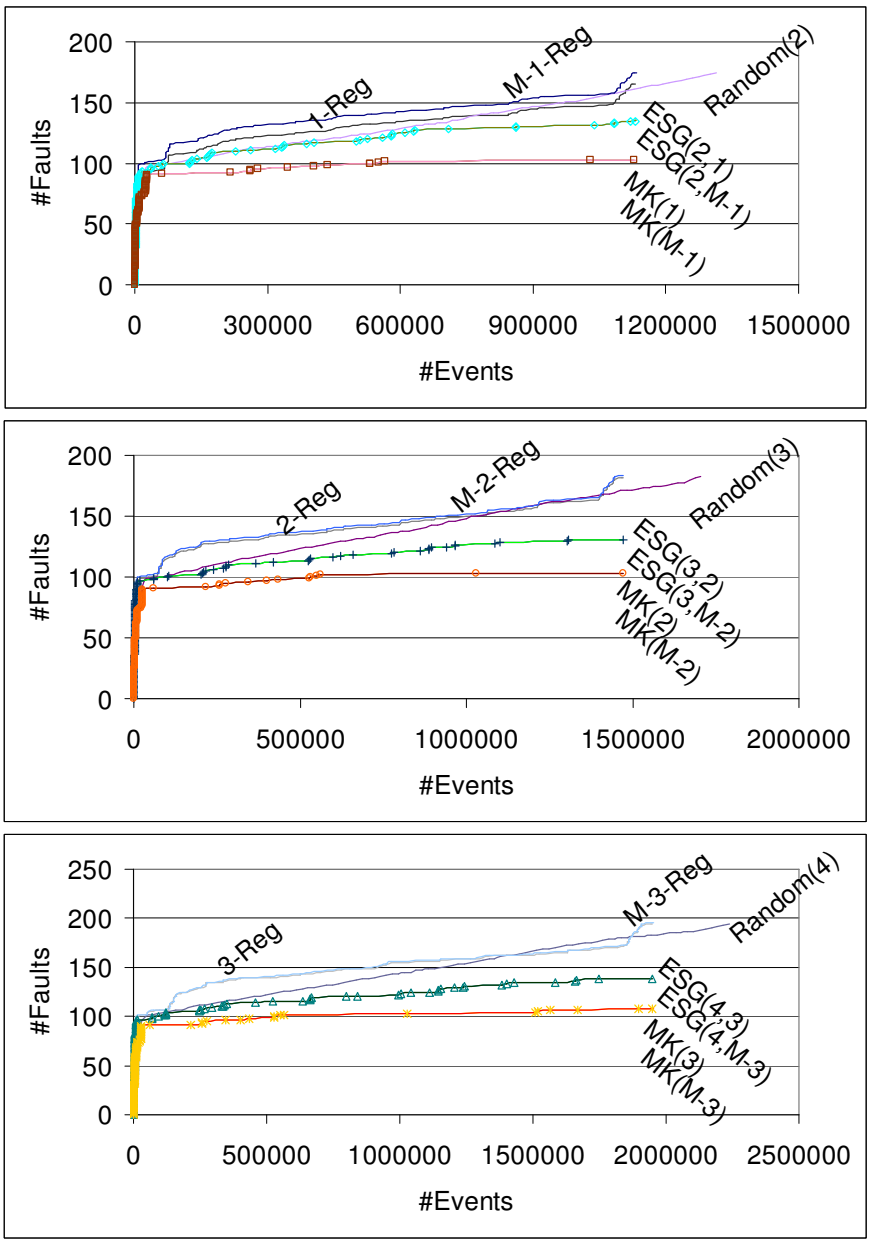


Figure B.1. Test execution curves for ShearBar.

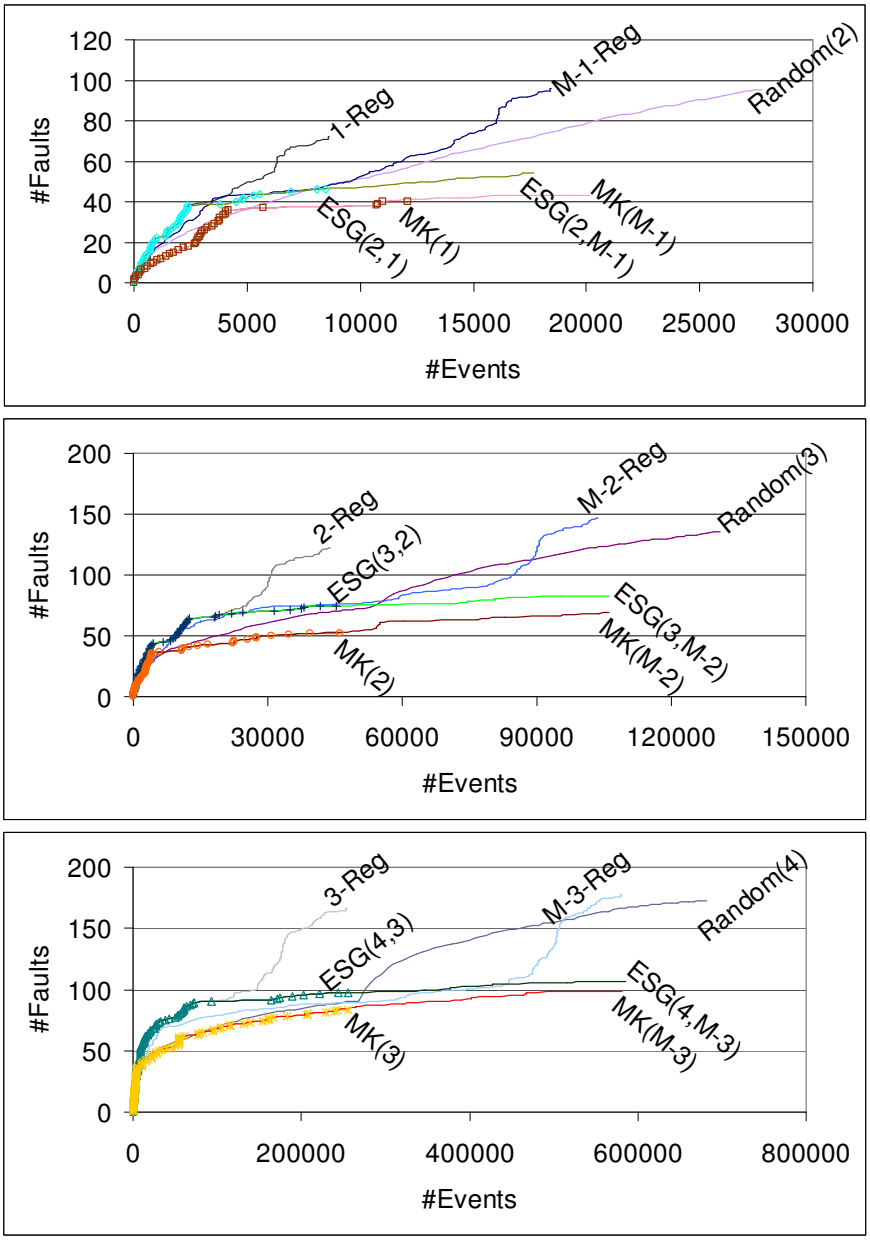


Figure B.2. Test execution curves for Specials.

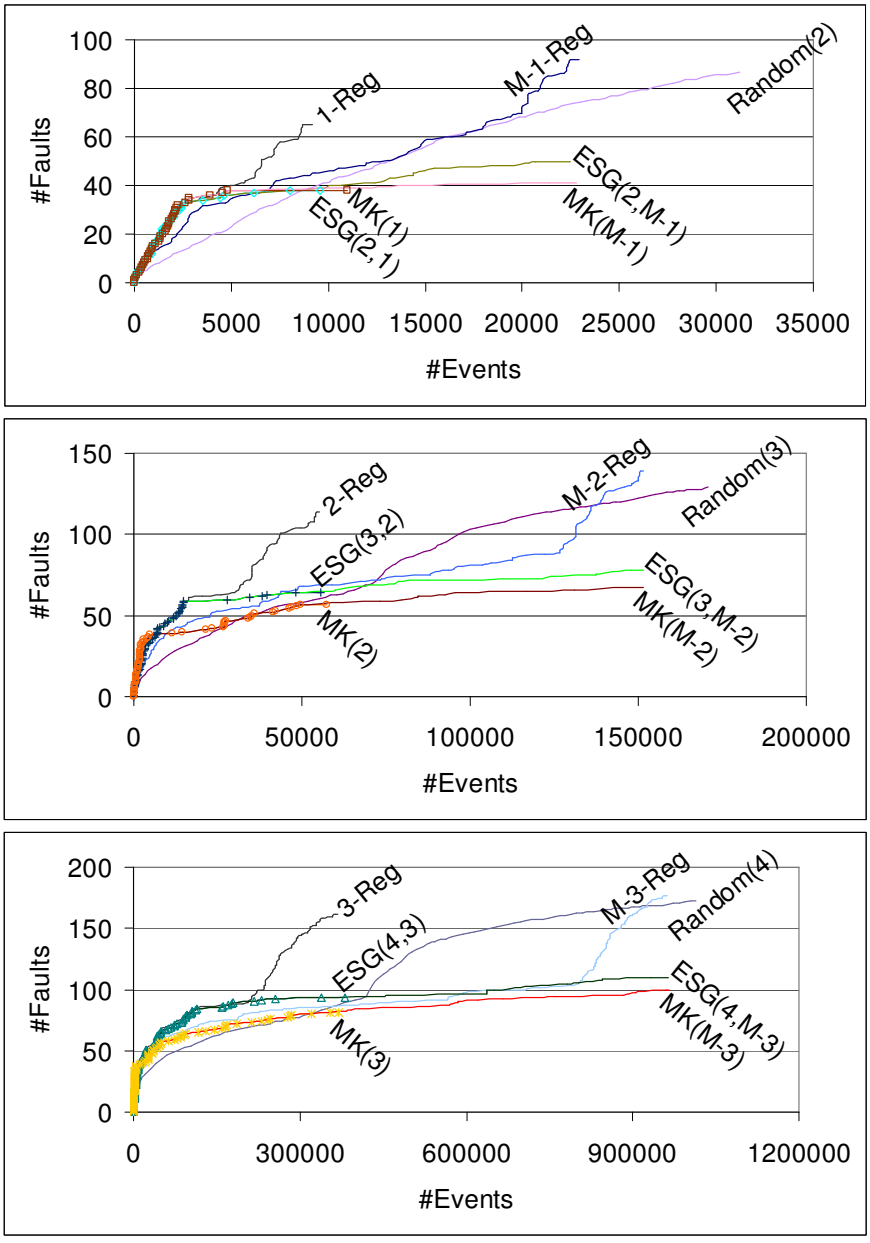


Figure B.3. Test execution curves for Additionals.