Running Title: Model-Based GUI Testing

Advances in Model-Based Testing of Graphical User Interfaces

FEVZİ BELLİ

Department of Electrical Engineering and Information Technology (EIM-E/ADT), University of Paderborn, D-33095, Paderborn Germany

MUTLU BEYAZIT

Department of Computer Engineering, Yaşar University, Üniversite Caddesi, No:37-39, Ağaçlı Yol, Bornova, İzmir, Turkey

CHRISTOF J. BUDNIK

Siemens Corporation, Corporate Technology, Princeton, USA
E-mail: christof.budnik@siemens.com

TUGKAN TUGLULAR

Department of Computer Engineering, Izmir Institute of Technology, 35430 Urla Izmir, Turkey

* Author names are in alphabetical order.

Abstract

Graphical user interfaces (GUIs) enable comfortable interactions of the computer-based systems with their environment. Large systems usually require complex GUIs which are commonly fault-prone and thus are to be carefully designed, implemented, and tested. As a thorough testing is not feasible, techniques are favored to test relevant features of the system under test that will be specifically modeled. This chapter summarizes, reviews, and exemplifies conventional and novel techniques for model-based GUI testing.

Keywords

Model-based testing, contract-based testing, graphical user interfaces, model morphology, test termination, test optimization, test automation

Outline

1 Introduction – User Interfaces and Their Holistic Testing

There are two distinct types of construction work while developing software:

- Design, implementation, and test of the programs.
- Design, implementation, and test of the user interface (UI).

We assume that UI might be constructed separately, as it requires different skills, and maybe different techniques than construction of common software. The design part of the development job requires a good understanding of user requirements; the implementation part requires familiarity with the technical equipment, i.e., programming platform, language, etc. Testing requires both: a good understanding of user requirements, and familiarity with the technical equipment. This chapter is about UI testing, i.e., testing of the software that realizes the UI, taking the design aspects into account. To some extent, also analysis aspects are covered because testing and analysis usually belong together.

Graphical user interfaces (GUIs) have become more and more popular and common UIs in computer-based systems. Testing GUIs is, on the other hand, a difficult and challenging task for many reasons: First, the input space possesses a great, potentially infinite number of combinations of inputs and events that occur as system outputs; external events may interact with these inputs. Second, even simple GUIs possess an enormous number of states which are also due to interaction with the inputs. Last but not least, many complex dependencies may hold between different states of a GUI system, and/or between its states and inputs.

User inputs are critical for the security, safety, and reliability of software systems. As Whittaker [1] indicated, "… data is the lifeblood of software; when it is corrupt, the software is as good as dead." According to Whittaker, this is indeed the bottom line for software developers and testers. One must consider every single input from every external resource to have confidence in the ability of the system under test (SUT) to properly handle malicious attacks and unanticipated operating environments. Deciding which inputs to trust and which to validate is a constant balancing act. Experiences from safety and security fields [1] have shown that user inputs, mostly obtained from GUIs, should be validated thoroughly to prevent attacks ranging from injection to denial of service and resulting in intrusion or even in system crashes. The same is true for safety violations.

Nevertheless, nowadays it is taken for granted that most Human-Computer Interfaces (HCI) are materialized via GUIs. There exists a vast amount of research work for specification of HCI, resulting in an effective testing strategy which is not only easy to apply, but also scalable in sense of stepwise increasing the test complexity and accordingly the test coverage and completeness of the test process, thus also stepwise increasing the test costs in accordance with the test budget of the project. There has been, however, little well known systematic study in this field. This chapter presents techniques to systematically test GUIs, being capable of test case enumeration for precise test scalability. Aspects of test optimization and rationalization by tools are also covered.

Test cases generally require the determination of meaningful test inputs and expected system outputs for these inputs. Accordingly, to generate test cases for a GUI, one has to identify the test objects and test objectives. The test objects are the instruments for the input, e.g. screens, windows, icons, menus, pointers, commands, function keys, alphanumerical keys, etc. The objective of a test is to generate the expected system behavior (desired event) as an output by means of well-defined test input, or inputs. In a broader sense, the test object is the software under test (SUT); the objective of the test is to gain confidence in the SUT. Robust systems possess also a good exception handling mechanism, i.e., they are responsive not in terms of behaving properly in case of correct, legal inputs, but also by behaving good-natured in case of illegal inputs, generating constructive warnings, or tentative correction trials, etc. that help to navigate the user to move in the right direction. In order to validate such robust behavior, one needs systematically generated erroneous inputs which would usually entail injection of undesired events into the SUT. Such events would usually transduce the software under test into an illegal state, causing even a system crash, if the program does not possess an appropriate exception handling mechanism. This is called "negative testing" and is also subject of this chapter.

Test inputs of a GUI usually represent sequences of GUI-object activities and/or selections that operate interactively with the objects (Interaction Sequences – IS, [2], see also [3] and [4]). Among these ISs, the ones interesting for testing are externally observable (Event Sequences – ES). Such an ES is complete (CES), if and only if it eventually invokes the desired system responsibility. From Knowledge Engineering point of the view, the testing of GUI represents a typical planning problem that can be solved goal-driven [4]: Given a set of operators, an initial state and a goal state, the planner is expected to produce a sequence of operators that change the initial state to the goal state. For the GUI test problem described above, this means we have to construct the test sequences in dependency of both the desired, correct events (positive testing) and the undesired, faulty events (negative testing). A major problem is the unique distinction between correct and faulty UI events (Oracle Problem, [5] and [6]). The chapter reviews approaches that exploit the concept of ES to elegantly cope with the Oracle Problem.

GUI testing can be performed using model-based testing (MBT) approach. In MBT, a model describing the behavior of SUT is created and this model is used for automatic generation of test cases which are then applied to the SUT. The basic idea is to use some coverage criteria to generate test cases [4], [5] and [6]. Achieving a proper level of coverage entails the generation of test cases and the selection of an optimal number of them. Thus, it ensures the cost-effective exercise of a given set of structural or functional features.

Another tough problem while testing is the decision when to stop testing (Test Termination Problem and Testability [5] and [6]). Exercising a set of test cases, the test results can be satisfactory, but this is limited to these special test cases. Thus, for the quality judgement of the SUT one needs further, rather quantitative arguments, usually materialized by well-defined coverage criteria. The most well-known coverage criteria are based either on special, structural issues of the software to be tested (implementation orientation/white-box testing), or its behavioral, functional description (specification orientation/black-box testing), or both, if both implementation and specification are available (hybrid/gray-box testing).

Putting the different components of the approach together, a holistic way of modeling of software development is materialized, with the novelty that the complementary view of the desired system behavior enables to obtain the precise and complete description of undesired situations, leading to a systematic, scalable, and complete fault modeling.

The present chapter summarizes existing work on model- and specification-based, positive and negative GUI testing, depicting it by many examples, from simple ones, e.g., copy/cut-paste process, to examples lent from public domain Internet, e.g., Real Jukebox (an interactive personal music management program), and to examples lent from real projects, e.g., from the automotive industry, namely a proactive system to control a marginal strip mower mounted on a truck.

This chapter prefers graph-based modeling technique as it is widely accepted in the practice. Moreover, formal methods, for example graph theory, can be applied to graph-based models for achieving algorithms for design, validation & verification, and optimization. All sections use, exemplarily, ESG modeling. For enabling quantification of test cases ESG modeling will be augmented by decision tables. Moreover, the idea of “model morphology” will be introduced from mutation analysis and testing to refine the holistic view.

Section 2 introduces the notion of finite-state modeling centered mainly on event-based models which are most commonly used both for modeling the system and the faults through sequence relation between the events. Notions and results from mutation analysis and testing have been adopted for test case generation. Cost aspects are discussed in Section 3 that introduces an optimization model to solve the test termination problem. Some potentials of test cost reduction are discussed. A new approach, GUI testing based on contracts and decision tables, is introduced in Section 4. Section 5 reviews existing tools for GUI testing, before Section 6 concludes the chapter, considering also future research directions.

Techniques represented in this chapter are to a great extend based on sound mathematical methods. Thus, they enable a formal, algorithmic handling, and consequently automatization.

The authors decided to integrate related work into the relevant sections of the chapter instead of having an extra section. This simplifies the work, and avoids multiple referencing, that is, once in relevant section(s), and another time in the extra section on related work.

2 Modeling of GUIs of Interactive Systems

In terms of behavioral patterns, the relationships between the system under test and its environment, i.e., the user, the natural environment, etc., can be described as proactive, reactive or interactive. In the case of pro-activity, the system generates the stimuli, which evoke the activity of its environment. In the case of reactivity, the system behavior evolves through its responses to stimuli generated by its environment. Most human-computer interfaces are nowadays interactive in the sense that the user and the system can be both pro- and reactive. In this particular kind of system, the user interface (UI) can have another environment, that is, the plant, the device or the equipment, which the UI is intended for and embedded in, controlling technical processes. Modern UIs will be mostly implemented graphically; therefore, we will concentrate on graphical, interactive user interfaces (GUI); UI and GUI will be used interchangeably.

This section discusses the modeling of GUIs for the purpose of testing. In Section 2.1, different GUI modeling techniques are introduced; the discussion is mainly centered around event-based models such as, event flow graphs (EFGs), event sequence graphs (ESGs) and k-sequence right regular grammars (k-Regs). In Section 2.2, Section 2.3, and Section 2.4, the discussion is based on a generic event-based modeling methodology using k-Regs. In Section 2.2, the basic notions and properties needed for testing purposes and fault-based aspects are analyzed. In Section 2.3, the concept of varying model morphology and its benefits are discussed and, in Section 2.4, test generation perspectives are demonstrated. Section 2 is concluded by Section 2.5 by laying out and discussing some issues related to the model-based testing methodologies.

2.1 **Different Techniques for Modeling GUIs**

An event is defined as an externally observable phenomenon, e.g., a user's, stimulus or a response of the GUI, punctuating different stages of the system activity. Event-based modeling techniques are commonly used for modeling GUIs. The most well-known event-based techniques are centered on the use of event sequence graphs (ESGs) [7], event flow graphs (EFGs) [8], and a particular type of regular grammars called k-sequence right regular grammars (k-Regs) [9][10].

Although we focus on event-based models, state-based or algebraic models can also be used for testing of GUIs. FSMs [4][11][12], variable FSMs [13], hierarchical FSMs [14], statecharts [15][16] and pushdown automata [17][18] are common examples to state-based models; whereas, regular expressions [19] can be considered as algebraic.

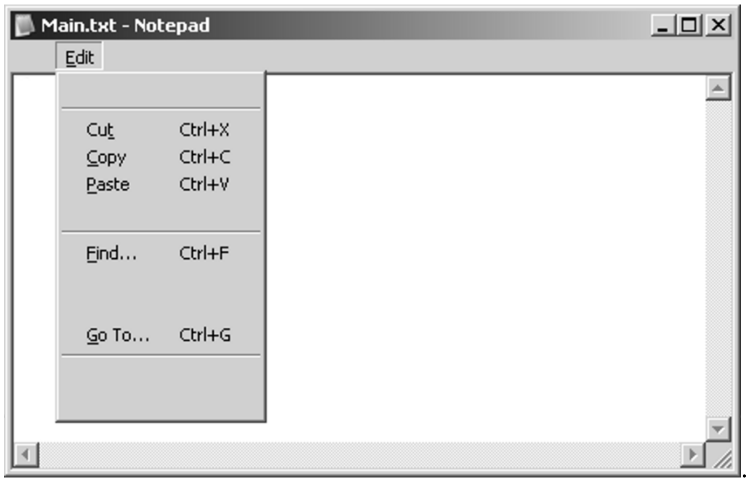
2.1.1 Event Sequence Graphs

An *event sequence graph* (*ESG*) is a 4-tuple *(N, A, S, F)* where

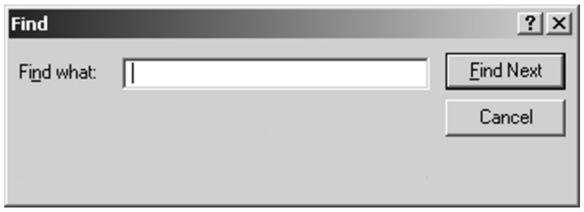
- N is a finite set of nodes representing the *events*.
- $A \subseteq N \times N$ is a finite set of directed arcs representing *follows* relation between events; that is, for two events x and y in the graph, x *follows* y if and only if (y, x) is an arc in the graph.
- $S \subseteq N$ is a distinguished nonempty set of events representing *start* or *initial* events.
- $F \subseteq N$ is a distinguished nonempty set of events representing *finish* or *final* events.

The above definition suggests that ESG-based modeling employs a very simplistic approach. Each node in an ESG is an event whose type or application specific semantics are ignored; and each arc represents a sequence of two events.

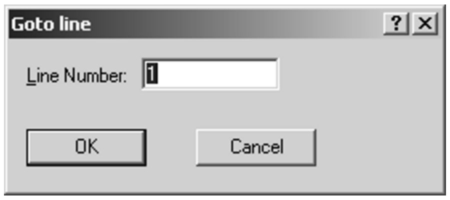
An example GUI is given in Figure 1. It contains a total of 11 events (all taskbar and non-GUI-related events are ignored): “Cut”, “Copy”, “Paste”, “Find…”, “Go To…”, “Find what”, “Find Next”, “Cancel (Find)”, “Line Number”, “OK” and “Cancel (GoTo)”. Performing “Go To…” and “Find…”events bring forth sub-components of the GUI enabling the execution of different events. After “Find…”, “Find what”, “Find next” and “Cancel (Find)” and, after “Go To…”, Line Number”, “OK” and “Cancel (GoTo)” events are enabled.



(a) “Main” window



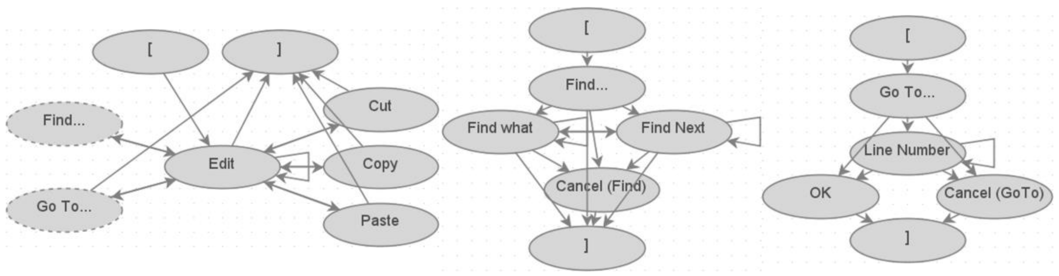
(b) “Find” window (modeless).



(c) “Goto line” window (modal).

Figure 1. An example GUI (Simplified from Notepad). [20]

Figure 2 is an ESG model of the GUI given in Figure 1. Events are only distinguished as simple and composite events: Simple events are shown in ellipses and they correspond to actual events. The composite events (“Go To…” and “Find…”), which have their own ESGs, are shown in the dotted ellipses. Note that the actual “Go To…” and “Find…” events are in fact in their corresponding sub-ESGs. In addition, in all ESGs, pseudo-events “[” and “]” are used to mark start and finish events, respectively; that is, start events follow “[“, and “]” follows each finish event.



(a) "Main" ESG. (b) "Find" sub-ESG. (c) "Goto line" sub-ESG.

Figure 2. A simplified ESG model of the GUI in Figure 1. [20]

Note that, although not suggested by the definition, some events in the example ESG are composite; they represent sub-ESGs. This is a quite commonly employed technique in practice to ease the modeling. Such ESGs are actually called *structured* ESGs [21][22] where certain nodes can be refined as long as the refinement is compatible with the notion of event. Similarly, it is possible to extend ESGs to enable the modeling of more complex situations as follows [21][22].

- *Input-output* ESGs: Input and output events are distinguished semantically.
- *Communicating* ESGs: Two ESGs can communicate with each other to accomplish a task.
- Quiescent ESGs: A special event to represent the occurrence of no input or output system action is also included.
- *Timed* ESGs: Event-based behavior is defined with respect to time.
- *Pushdown* ESGs: A stack component is included as a special type of memory.

2.1.2 Event Flow Graphs

An *event flow graph* (*EFG*) [8], on the other hand, takes component-based structure of the GUIs into account and distinguishes between different types of events such as menu-open events, restricted-focus events, unrestricted focus events, termination events and system-interaction events [23]. In this perspective, EFGs can be considered as an extension to ESGs where node semantics are augmented using application specific details. An EFG for a GUI component C is a 4-tuple (V, E, B, I) where

- V is a set of vertices representing all the *events* in the component. Each $v \in V$ represents an event in C where it can be a menu-open, a restricted-focus, a termination or a system-interaction event.
- $E \subseteq N \times N$ is a set of *directed edges* between vertices. We say that event e_i follows e_j if and only if e_j may be performed *immediately* after e_i. An edge $(v_x, v_y) \in E$ if and only if the event represented by v_y *follows* the event represented by v_x.
- $B \subseteq V$ is a set of vertices representing those events of C that are available to the user when the component is first invoked.
- $I \subseteq V$ is the set of restricted-focus events of the component.

Figure 3 shows an example EFG model of the GUI in Figure 1. The events are distinguished based on their types and different shapes are used for such events in modeling. "Edit" is a menu-open event (shown in diamond), "Go To…" is a restricted-focus event (shown in double circle), and related "OK" and "Cancel (GoTo)" are termination events (shown in rectangles). Also, "Find…" is an unrestricted focus event (because "Find" window is modeless) enabling "Find what" and "Cancel (Find) events, and "Cut", "Copy" and "Paste" events are system-interaction events (shown in circles). Furthermore, "Edit" is designated as the only start event.

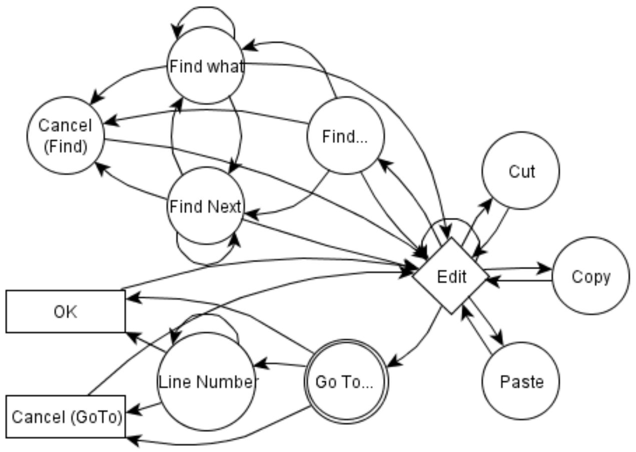


Figure 3. A simplified EFG model of the GUI in Figure 1. [20]

EFGs also have different extensions.

- Event Interaction Graphs (EIGs) [24]: An EIG focuses on system interaction and termination events (assuming that other events are not fault-prone), and interactions between them.
- Event Semantic Interaction Graphs (ESIGs) [25]: An ESIG models a subset of *follows* relation between events that are shown to interact at a certain semantic level.
- Probabilistic EFGs (PEFGs) [26]: In a PEFG, by analyzing usage profiles, weights or probabilities are assigned to the events to form Bayesian networks and n-gram Markov models.

2.1.3 k-sequence Right Regular Grammars

In order to model the relations between sequences of events and single events, a generalized family of event-based models is introduced as *k-sequence right regular grammars (k-Regs) (k≥1)* [27][9][10]. Mainly, k-Regs are defined with a general event-based modeling methodology in mind (although certain elements in the model can also be regarded as states, and, thus, enables adoption of state-based approaches). Therefore, event semantics are not specialized using application specific details. Furthermore, certain ESG-based formalization issues are solved resulting in improvements in the event-based testing approaches. A *k-Reg (k≥1)* is a 6-tuple *(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*. For each k-sequence $r \in K$, $r = r_1 \dots r_k$ and $d(r) = d(r_1) \dots d(r_k) \in B^k$ is the corresponding *basis k-sequence*.
- C is a finite set of *contexts* where $S \in C$ is the *start context*.
- P is a finite set of *productions* of the form

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

  where $Q \in C$ is a context, $r \in K$ is a k-sequence, $c(r) \in C \backslash \{S\}$ is the unique context of r, and ε is the empty string. If $k \geq 2$, for each $c(q) \to r\ c(r) \in P$, the ending (k-1)-sequence of q is the beginning (k-1)-sequence of r.

The semantics of the productions of a k-Reg G is as follows.

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

Figure 4 shows an example 1-Reg model of the GUI in Figure 1. Productions of the form $S \to r\ c(r)$ and $c(r) \to \varepsilon$ are used to mark start events and finish events, respectively. Furthermore, productions of the form $c(q) \to r\ c(r)$ are used to model the *follows* relation between single events. Therefore, it is possible to represents the productions visually using directed graphs (similar to ESGs).

1. S → Edit c(Edit)
2. c(Edit) → ε
3. c(Edit) → Edit c(Edit)
4. c(Edit) → Cut c(Cut)
5. c(Edit) → Copy c(Copy)
6. c(Edit) → Paste c(Paste)
7. c(Edit) → GoTo... c(GoTo...)
8. c(Edit) → Find... c(Find...)
9. c(Cut) → ε
10. c(Cut) → Edit c(Edit)
11. c(Copy) → ε
12. c(Copy) → Edit c(Edit)
13. c(Paste) → ε
14. c(Paste) → Edit c(Edit)
15. c(GoTo...) → LineNumber c(LineNumber)
16. c(GoTo...) → OK c(OK)
17. c(GoTo...) → Cancel(GoTo) c(Cancel(GoTo))
18. c(LineNumber) → LineNumber c(LineNumber)
19. c(LineNumber) → OK c(OK)
20. c(LineNumber) → Cancel(GoTo) c(Cancel(GoTo))
21. c(OK) → ε
22. c(OK) → Edit c(Edit)
23. c(Cancel(GoTo)) → ε
24. c(Cancel(GoTo)) → Edit c(Edit)
25. c(Find...) → Edit c(Edit)
26. c(Find...) → Findwhat c(Findwhat)
27. c(Find...) → FindNext c(FindNext)
28. c(Find...) → Cancel(Find) c(Cancel(Find))
29. c(Findwhat) → Edit c(Edit)
30. c(Findwhat) → Findwhat c(Findwhat)
31. c(Findwhat) → FindNext c(FindNext)
32. c(Findwhat) → Cancel(Find) c(Cancel(Find))
33. c(FindNext) → Edit c(Edit)
34. c(FindNext) → Findwhat c(Findwhat)
35. c(FindNext) → FindNext c(FindNext)
36. c(FindNext) → Cancel(Find) c(Cancel(Find))
37. c(Cancel(Find)) → Edit c(Edit)

Figure 4. A simplified 1-Reg model of the GUI in Figure 1.

ESGs, EFGs and k-Regs are all used to define a *follows* relation by properly identifying start and finish events. In case of ESGs and EFGs, this relation is between events; however, in case of k-Regs, this relation is between k-sequences and events. From a formal point of view, ESGs, EFGs and 1-Regs are similar to FSMs. By loosening the constraints on start and finish events, one can convert an ESG, an EFG or a 1-Reg to an FSM by interpreting the formers as Moore-like machines [28] and the latter as Mealy-like machine [29], and vice versa. However, the case of empty string and the absence of final states should be handled carefully. Furthermore, use of indexing (or contexting) [7] may be necessary to assign a unique label to each event.

2.2 Analysis of Models

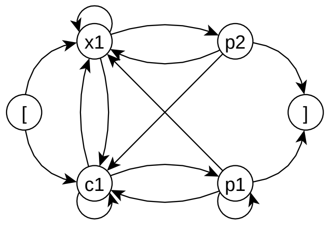
In general, when a model is given, different types of analyses can be carried out. Considering the aforementioned event-based models, for event-based testing of GUIs, it is possible to perform an analysis to test for *legal* (*valid, desirable*, *correct* or *positive*) and *illegal* (*invalid, undesirable*, *faulty* or *negative*) *system behaviors*, that is, the behaviors allowed and not allowed by the system, respectively. In this section, in order to represent such behaviors and carry out our analysis, the discussion is based on certain notions borrowed from mutation analysis and testing [30][31][32].

2.2.1 Basic Notions

Let *G* be a k-Reg. Event sequences that can and cannot be derived using *G* are distinguished for testing. An event sequence *s* is said to be *in G*, if it can be derived using the productions (or the *follows* relation) in *G*. A nonempty event sequence *s* in *G* is a *start sequence*, if it starts with a start event; *s* in *G* is a *finish sequence*, if it ends with a finish event.

Figure 5 shows an example 1-Reg where *c* is *copy*, *x* is *cut* and *p* is *paste* basis event, and *c1, x1, p1* and *p2* are contexted versions of these events. *{c1 x1, c1 p1 p1, p2 x1 p2, p1 p1 p1 c1}* is an example set which contains some sequences in this 1-Reg; whereas, *{c1 p2, x1 p1 p2, p2 p2 x1 c1}* contains some sequences not in it. Note that, in the figure, productions of the form $H \rightarrow T1$, $H \rightarrow T2$, …, $H \rightarrow TN$ are grouped as $H \rightarrow T1 \mid T2 \mid \ldots \mid TN$ to save space.

$$
\begin{aligned}
&S \rightarrow c1\ c(c1) \mid x1\ c(x1)\\
&c(c1) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid p1\ c(p1)\\
&c(x1) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid p2\ c(p2)\\
&c(p1) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid p1\ c(p1) \mid \varepsilon\\
&c(p2) \rightarrow c1\ c(c1) \mid x1\ c(x1) \mid \varepsilon
\end{aligned}
$$



(a) Productions. (b) Directed graph visualization.

Figure 5. A 1-Reg model. [10]

An event sequence in *G* can be used to exercise some desirable or correct behavior; whereas, an event sequence not in *G* can be used to exercise some undesirable or faulty behavior and it is also called as a *faulty event sequence (FES)*. Original model *G* and its mutants can be used to generate positive and negative test cases for these purposes. More precisely, 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.

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). $T_{CES}(G) \subseteq T_P(G)$ denotes the *set of all CESs*. Furthermore, an event sequence is a *negative test case*, if the first event in it is a nonstart event or it contains at least one 2-sequence which is not in M. $T_N(G)$ denotes the *set of all negative test cases*. A *faulty complete event sequence (FCES)* is a negative test case which either is composed of only a nonstart event, or contains only one 2-sequence which is not in *G* and it ends with this 2-sequence. $T_{FCES}(G) \subseteq T_N(G)$ denotes the *set of all FCESs*.

2.2.2 Relevant Mutants

In general, one can create infinitely many mutants modeling multiple missing event or extra event faults. For this purpose, *marking* (*mark start*, *mark finish*, *mark non-start* and *mark non-finish*), *insertion* (*insert sequence, insert k-sequence*) and *omission* (*omit sequence* and *omit k-sequence*) operators can be defined [33] by extending the operators defined in [34]. In the light of the following assumptions which are generally valid for GUI testing, the types and the numbers of mutants can be reduced greatly [9][10].

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 sequence can be any event; a test sequence needs not to end with a finish event.

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

P1. Missing and extra event faults are limited by considering the k-sequences 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 k-sequence mutants are discarded; they do not contain any k-sequence that is not contained in the original model. (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 k-sequence mutants. (by definition [33])

P5. 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)

Consequently, we do not need to use all types of mutants for test generation; we can use the original k-Reg to cover (k+1)-sequences to reveal missing event faults, and mark start and insert k-sequence mutants to cover faulty (k+1)-sequences to reveal extra event faults. Basically, the purpose of using a mark start operator is to turn a given k-sequence into a start k-sequence. In this way, mutant models which have extra start k-sequences can be constructed. These models are used to generate test cases to reveal extra event faults where the extra event is a start event in a start k-sequence. The operator is defined as follows.

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

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

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

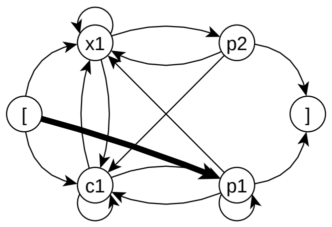
Insert k-sequence operator adds a new k-sequence to a given grammar following an existing k-sequence. In this way, mutant models which contain k-sequences with different contexts can be created. Such models are used to generate test cases to reveal extra event faults where the extra event follows a k-sequence. The operator is defined as follows.

Given a k-Reg $G = (E, B, K, C, S, P)$, a k-sequence e such that $e \notin K$ and $d(e) \in B^k$, and an existing k-sequence $a \in K$, *insert k-sequence (It)* operator is defined as

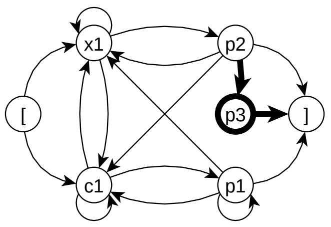
$$It(G, e, a) = G' = (E, B, K', C', S, P'),$$

where $K' = K \cup \{e\}$, $C' = C \cup \{c(e)\}$, and $P' = P \cup \{c(a) \rightarrow e\, c(e), c(e) \rightarrow \varepsilon\}$.

Figure 6 shows a mark start and an insert 1-sequence mutant of the 1-Reg in Figure 5 (To save space, only the directed graph visualizations are included).



(a) *p1* is marked as start.



(b) *p3* (a *p* event) is inserted after *p2*.

Figure 6. A mark start and an insert 1-sequence mutant of the 1-Reg in Figure 5. [10]

2.2.3 Mutant Selection

In event-based testing (under assumptions A1 and A2), not all mutants need to be generated. Therefore, having defined the relevant mutation operators, two strategies are defined for mark start and insert k-sequence mutants to select a subset of all possible mutants in such a way that

- 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 test cases to reveal the faults,
- the generation of equivalent mutants and multiple mutants modeling the same faults are avoided, and
- a test case to reveal the fault modeled by the mutant can be generated in linear time.

Given a k-Reg $G = (E, B, K, C, S, P)$, mark start and insert k-sequence mutant selection strategies are defined as follows.

Mark Start Mutant Selection Strategy: 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 $d(y_1) = d(e_1)$.

Insert k-sequence Mutant Selection Strategy: For each insert k-sequence mutant $It(G, e, a)$ of G, k-sequence e and k-sequence a are selected as a mutation parameter if the following conditions hold:

1. There exists no $c(a) \rightarrow x\ c(x) \in P$ such that $d(x_k) = d(e_k)$.
2. There exists no previously selected mutation parameter (y, a) such that $d(y_k) = d(e_k)$.

The algorithms to generate all mark start and insert k-sequence mutants according to the above strategies are given in Algorithm 1 and Algorithm 2, respectively.

Algorithm 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
  $M = \emptyset$, $N = \emptyset$
  for each $b \in B$ do
      if there is no $S \rightarrow x\ c(x) \in P$ such that $d(x_1) = b$ and
         there is no $y \in N$ such that $d(y_1) = b$ then
           Select a k-sequence $e \in K$ such that $d(e_1) = b$
           $G' = G$
           $M = M \cup \{Ms(G', e)\}$
           $N = N \cup \{e\}$
      endif
  endfor
```

Algorithm 2. Insert k-sequence Mutant Selection

```
Input: $G = (E, B, K, C, S, P)$ – the input grammar
Output: $M$ – the set of selected insert k-sequence mutants
  $M = \emptyset, N = \emptyset$
  for each $a \in K$ do
      $N = \emptyset$
      for each $b \in B$ do
          if there is no $c(a) \rightarrow x\ c(x) \in P$ such that $d(x_k) = b$ and
            there is no $(a, y) \in N$ such that $d(y_k) = b$ then
              $b'$ = a new contexted version of $b$, $e = a_2 \dots a_k\ b'$
              $G' = G$
              $M = M \cup \{It(G', e, a)\}$,
              $N = N \cup \{(a, e)\}$
          endif
      endfor
  endfor
```

Let *G* be the 1-Reg in Figure 5. 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)*. Furthermore, one can only use basis 1-sequence *p* to generate insert 1-sequence mutant, because c and x can follow all events. The only selected insert 1-sequence mutant is *It(G, p3, p2)*, because only *p2* is not followed by a *p* event.

2.3 Exploiting Model Morphology for Event-Based Testing

Varying model morphology can be formalized using k-Regs [9][10]. However, not to bore the reader with too much formalism, we keep the discussion semi-formal and skip certain details. Interested reader may refer [10][33] for a more complete discussion.

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. For this purpose, a transformation to vary *k* and generate models with morphological differences is defined as follows.

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
  $$G_{k+1} = (E, B, K_{k+1}, C_{k+1}, S, P_{k+1}) \text{ where}$$
    - $K_{k+1} = \{q_1 \dots q_k\ r_k \mid c(q) \rightarrow r\ c(r) \in P_k$ 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.
    - $C_{k+1} = \{c(r) \mid r \in K_{k+1}\}$ is the set of contexts.
    - $P_{k+1} = \{S \rightarrow r\ e\ c(r\ e) \mid S \rightarrow r\ c(r) \in P_k$ and $c(r_k) \rightarrow e\ c(e) \in P_1\} \cup$
      $\{c(q\ r_k) \rightarrow \varepsilon \mid c(q) \rightarrow r\ c(r) \in P_k$ and $c(r_k) \rightarrow \varepsilon \in P_1\} \cup$
      $\{c(q\ r_k) \rightarrow r\ e\ c(r\ e) \mid c(q) \rightarrow r\ c(r) \in P_k$ and $c(r_k) \rightarrow e\ c(e) \in P_1\}$ is the set of productions.

The above definition is recursive; it can easily be converted to an iterative algorithm whose steps are outlined in Algorithm 3.

Algorithm 3. 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
```

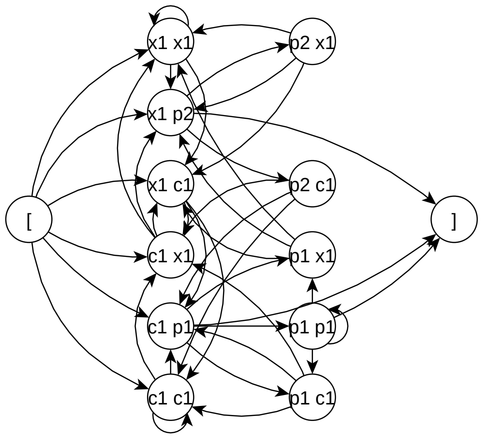
Based on the definition, Algorithm 3 uses a 1-Reg and a k-Reg to obtain a (k+1)-Reg. Basically, what happens is as follows.

- 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.
- 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.

Figure 7 is the corresponding 2-Reg transformed from the 1-Reg in Figure 5.

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.

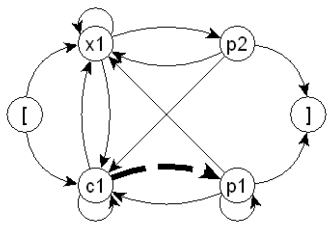


(b) Directed graph visualization.
Figure 7. A 2-Reg model (Transformed from Figure 5). [10]

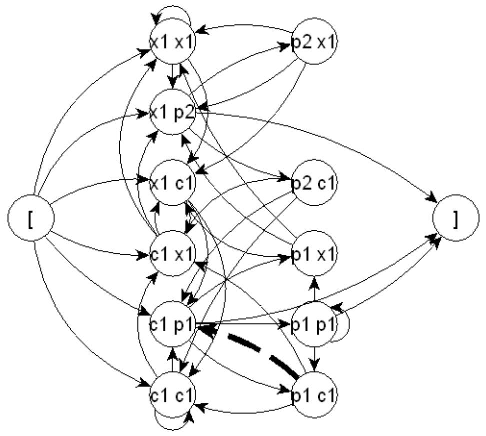
One of the most important benefits of using of morphologically different models, generated using grammar transformation, is the extension of the set of possible mutants (or fault models). To see this, consider the mutant of Figure 7 generated by omitting sequence *(p1 c1, c1 p1)* as shown in Figure 8b. This mutant models the fault that

p1 is missing after *p1 c1*;

that is, *p* fails after performing a *p* and a *c*. It is not possible to create such a mutant from the model in Figure 5 by a simple omission. For example, one can omit sequence *(c1, p1)* (See Figure 8a). However, in this mutant, *p* fails immediately after performing a *c*. Hence, the mutant in Figure 8b models a different and more subtle fault than the mutant in Figure 8a. Thus, the set of fault models can be extended by generating mutants modeling different or more subtle faults.



(a) A mutant of the model in Figure 5.



(b) A mutant of the model in Figure 7.

Figure 8. Two mutants generated from morphologically different models (Mutations are shown in boldface dashed lines). [10]

Actually, this example hints that using morphologically different models in test generation is also beneficial since it helps the detection of different or more subtle faults.

2.4 Test Generation from Models

Although it is possible to use different coverage criteria for test generation considering different (even application specific) semantics, such as inter-component coverage criterion [8] which utilizes the structure of the GUIs, we limit our discussion for test generation to more generic k-sequence and faulty k-sequence coverage criteria for detection of missing event and extra event faults as discussed in Section 2.2.

Given an event-based model *G* and a set of sequences *X*. *X* is said to cover a k-sequence *r* in *M*, if *r* appears in a sequence in *X*, and, if *X* covers all k-sequences in *G*, it is said to achieve *k-sequence coverage*. Furthermore, *X* is said to cover a faulty k-sequence *r* which is not in *G*, if *r* appears in a sequence in *X*, and, if *X* covers all k-sequences not in *G*, it is said to achieve *faulty k-sequence coverage*.

As discussed in Section 2.2, k-sequence coverage is used to reveal missing event faults where an event does not follow a (possibly empty) sequence of events. 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 $k \geq 1$. 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 addition, 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 event. 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 a (possibly empty) sequence of events (although it should not). For this reason, we only consider the faulty k-sequences whose last events are faulty.

To satisfy these criteria, we use k-Reg models. However, there is a problem: A sequence *s* in the corresponding k-Reg G_k of a 1-Reg G_1 is not always a sequence in G_1. Therefore, in order to obtain a sequence *t* in G_1 from a sequence *s* in G_k, the following transformation is defined.

Let $s = u^1 \ldots u^m$ 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 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 example, $s = c1\ c1\ c1\ x1\ x1\ x1\ x1\ p2$ is a sequence in the 2-Reg in Figure 7 but it is not in the 1-Reg in Figure 5. However, $T_S^{-1}(s, 2) = c1\ c1\ x1\ x1\ p2$ is a 5-derived sequence in the 1-Reg in Figure 5.

2.4.1 Positive Test Generation

In order to generate test cases achieving (k+1)-sequence coverage from a given 1-Reg, its corresponding k-Reg can be used. In this way, one can reveal missing event faults where an event does not follow a certain k-sequence.

As discussed in Section 2.1, productions of a k-Reg encodes (k+1)-sequences in the system. Hence, by covering these productions, one can generate test sets achieving (k+1)-sequence coverage.

Algorithm 4. Test Generation to Achieve (k+1)-sequence Coverage

```
Input: $G = (E, B, K, C, S, P)$ – the input 1-Reg
       $k$ – an integer $\geq 1$
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 3 in Section 2.3
  $Y$ = 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 Section 2.4
  endfor
```

Algorithm 4 outlines the generation of a test set achieving (k+1)-sequence coverage from a given 1-Reg. In the algorithm, generating a test set achieving production coverage for the corresponding k-Reg model can be performed in different ways. For example, it is possible to perform some optimizations by adapting algorithms to solve Chinese Postman Problem over directed graphs, like [35][36][37], 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 [38][20]. Thus, algorithms such as those in [39][40][41] can also be used to generate relatively short but generally nonoptimized sequences from a given grammar, while using less resources.

When Algorithm 4 is executed on the 1-Reg in Figure 5 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 7, 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 basis events represent the events as they are visible to user.

2.4.2 Negative Test Generation

In order to generate test sets achieving fault (k+1)-sequence coverage, certain k-Reg mutants need to be generated and used. Therefore, before laying out the test generation algorithm, we discuss two mutant selection strategies based on mark start and insert k-sequence mutants (A detailed discussion on other possible k-Reg mutants and on the reasons for the selection of only these two types of mutants can be found in [33]).Having discussed mutant-related background, we can now proceed to the test generation algorithm to achieve faulty (k+1)-sequence coverage as demonstrated in Algorithm 5.

Algorithm 5. Test Generation to Achieve Faulty (k+1)-sequence Coverage

```
Input: $G = (E, B, K, C, S, P)$ – the input 1-Reg
       $k$ – an integer $\geq 1$
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 3 in Section 2.3
  for each $G' = Ms(G_k, e)$ selected using Algorithm 1 in Section 2.2.3 do
      $X = X \cup \{e_1\}$
  endfor
  for each $G' = It(G_k, a, e)$ selected using Algorithm 2 in Section 2.2.3 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 Section 2.4
  endfor
```

When Algorithm 5 is executed on the 1-Reg in Figure 5 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.

2.5 Issues to Consider

There are certain issues related to the model-based GUI testing methodologies discussed so far in Section 2 which are worth mentioning.

Model Semantics: The models which are most commonly used for GUI testing have very simple semantics. They are mainly based on follows relation between events. This kind of abstractions may not be sufficient to capture the relevant behavior of certain applications.

Fault and Coverage Semantics: Being partially related to the semantics of the model, fault models and coverage criteria are relatively general. In case one is interested in only a specific fault type, the employed fault models and coverage criteria may be more than needed causing a waste of resources.

Size Complexity of Morphology Variation: In theory, the proposed morphology variation technique causes an exponential increase in the model size. Although, in practice, the value of *k* is almost always bounded, it still poses a limit by preventing the use of relatively larger *k* values.

Linear Test Cases: In most cases, generated test cases have a linear structure and they are composed of input events. This does not allow the possibility of a change in the flow of execution depending on responses from the system.

Order of Test Cases: Algorithms to generate test cases do not impose a specific order on the test cases. However, in practice, the fault detection efficiency of an approach may change depending on the order of executed test cases.

3 Testing and Test Optimization Exemplified by GUI-Modeling with ESG

Model-based GUI testing establishes a model that is used for guiding the test process to generate and select test cases, which form *sets* of test cases, or test *suites*. The selection is ruled by an *adequacy criterion,* which provides a measure of how effective a given set of test cases is in terms of its potential to reveal faults. Most model-based approaches use *coverage-oriented* adequacy criteria which determine how well the generated test suites cover the corresponding model. The ratio of the portion of the specification or code that is covered by the given test set in relation to the uncovered portion can then be used as a decisive factor in determining the point in time at which to stop testing (*test termination*).

A. Memon et al. introduced an approach to testing graphical UI (GUI) [42]. It deploys meth-ods of knowledge engineering to generate test cases, test oracles, etc., and to deal with the test termination problem. The approach uses some heuristic methods to cope with the state explosion problem which has the disadvantage that there is no guarantee that the best solution found is the optimal solution.

M. Marré, A. Bertolino [43] adopted the notion of "spanning set", which is similar to what has been introduced as "minimal spanning set of complete test sequences". A. Gargantini and C. Heitmeyer used a state-oriented approach [44], which is based on the traditional SCR (Software Cost Reduction) method. The approach uses model checking to generate test cases automatically from formal requirements specifications, using coverage metrics for test case selection. The approach is limited by the state space explosion problem.

Using the results of aforementioned approaches, and based on ESG notation, (Section 2), this section introduces a different method for GUI test optimization [45].

3.1 **Test Termination as an Optimizing Problem**

Coverage-oriented test termination must be, of course, economical in terms of an efficient test suite that is generated by optimizing the test execution time. Test execution time accounts for a significant portion of the overall test execution costs. Tests that are generated according to the coverage adequacy criteria are mostly too expensive to be executed as they require longer execution time, which leads to an efficiency deficit. Test sets need to be specifically structured to optimize the test execution time. The number and length of test cases of a test set are the primary factors that influence the cost of the test execution time in an automated test framework which does not need further human intervention or effort [45].

3.1.1 Minimizing the Test Sets of ESGs

As defined in Section 2, subsequent nodes traversed through an ESG represent an event sequence (ES). Sequences of length two are called *event pairs* (EPs). An event sequence is *complete* (CES) if the sequence includes the start and finish node and is also called a *walk* in the following. The union of the sets of CESs of minimal total length to cover the ESs of a required length is called *Minimal Spanning Set of Complete Event Sequences* (MSCES). If a CES contains all EPs at least once, it is called an *entire walk*. A legal entire walk is minimal if its length cannot be reduced. A minimal legal walk is *ideal* if it contains all EPs exactly once. Legal walks can easily be generated for a given ESG as CESs, respectively. It is not, however, always feasible to construct an entire walk or an ideal walk. Using some results of the graph theory [46], MSCESs can be constructed as the next section illustrates.

3.1.2 Minimal Spanning Sets of Complete Event Sequences

As mentioned in Section 2, a CES represents a *legal* walk, traversing the ESG from its entry to the exit. Given an ESG e, a complete legal walk contains each EP in e at least once. A complete legal walk is *minimal* if its length cannot be reduced without changing it to an incomplete legal walk. A minimal legal walk is considered *ideal* when it contains every EP exactly once. Legal walks can be generated easily for a given ESG as CESs. It is not, however, always feasible to construct a complete or an ideal walk. Using results from graph theory [46], MSCESs can be constructed as follows:

- Check whether an ideal walk exists.
- If not, check whether a complete walk exists and, if so, construct a minimal one.
- If there is no complete walk, construct a set of walks such that (a) sum of the lengths of all walks is minimal, and (b) all EPs are covered.

The MSCES problem introduced here has a lower degree of runtime complexity than the *Chinese Postman Problem* as the edges of the ESG are not weighted, i.e., the adjacent nodes are equidistant. In the following we summarize results relevant to the calculation of test costs that make the test process scalable. An algorithm described in [47] to solve the CPP determines a minimal tour that covers the edges of a given strongly connected graph. Transformation of an ESG into a strongly connected graph is illustrated in Figure 9. Addition of a backward edge, indicated as a dashed arrow from the exit to the entry, transforms the ESG in Figure 9 (a) to a strongly connected graph in Figure 9(b).

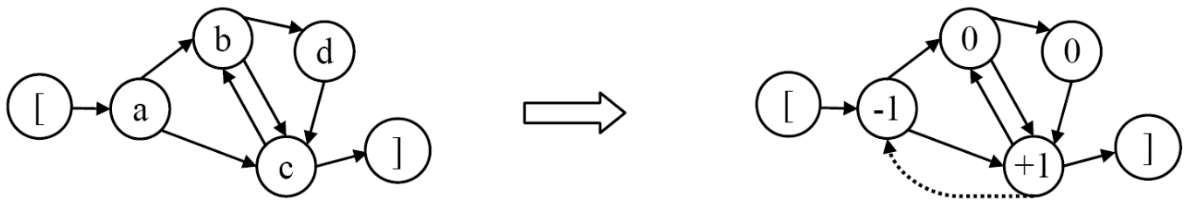


Figure 9 (a). An example ESG **(b).** Transferring walks into tours and balancing the node

The labels of the vertices in Figure 9 (b) indicate the *balance* of these vertices as the difference between the number of incoming edges and the number of the outgoing edges. These balance values determine the number of additional edges that will be identified by searching all-shortest-paths and solving the optimization problem. The problem can then be transformed into the construction of an *Euler tour* for this graph [46]. This tour may have multiple occurrences of the backward edge indicating the number of walks. For the ESG in Figure 9 (b), based on Figure 9 (a), the minimal set of the legal walks covering the EPs are **MSCES =** ***{abcbdc, ac}***. Note that no complete walks exist. Therefore, an ideal walk cannot be constructed.

Algorithm 6 calculates the MSCES for a given ESG as input. ε denotes the entry of the ESG and γ its exit. Given an event $v \in V$, diff(v) denotes the number of predecessor events of v minus the number of its successor events, which enables the construction of the *bags* (or *multisets*) A, B in the FOR-loop. We introduce the notation $\llbracket\,\rrbracket$ for *bags* and $\uplus$ *bag union*. They can be defined informally as follows. For instance, if diff(v)=3 in the first iteration step, assuming that A is initially empty, the bag A will consist of three instances of *v*, i.e., A = $\llbracket v, v, v \rrbracket$ after the assignment there. Note that $\llbracket v, v, v \rrbracket \neq \{v\}$, because the two entities on either side of the inequality sign ≠ are of different types; on the left-hand side is a bag (with three instances of *v*), whereas on the right-hand side is a singleton set with one element *v*. Turning to $\uplus$, note that $\llbracket v, v, v \rrbracket \uplus \llbracket v \rrbracket = \llbracket v, v, v, v \rrbracket$.

Algorithm 6. Generation of MSCES [51]

```
Input: ESG = (V, E, Ξ, Γ ); ε=[, γ=]
Output: MSCES
add_arc(ESG, (γ, ε));
bags A, B, M = ⟦⟧ ; set MSCES = ∅;                              //empty bags & set
FOR all nodes v∈V DO
    IF (diff(v) > 0) THEN FOR i:=1 TO diff(v) DO A = A ⊎ ⟦v⟧ ;
    IF (diff(v) < 0) THEN FOR i:=1 TO diff(v) DO B = B ⊎ ⟦v⟧ ;
m = |A| = |B|;
        //cardinality
D[1 .. m][1 .. m];
        //distance matrix D
FOR all nodes v∈A DO
    compute_shortest_paths(v, B, D);
M = solveAssignmentProblem(D);
FOR all (i, j)∈M DO
    Path = get_shortest_path(i, j);
    FOR all arcs e∈Path DO
        add_arc(ESG, e);
EulerTourList = compute_Euler_tour(ESG);
start = 1;
FOR i=2 TO length(EulerTourList)-1
    IF (getElement(EulerTourList, i) = γ) THEN
        MSCES = MSCES ∪ getPartialList(EulerTourList, start, i);
  start = i + 1;
RETURN MSCES;
```

3.1.3 *Minimal Spanning Set of Faulty Complete Event Sequences*

The concatenation of an event sequence ES and a faulty event pair (FEP), i.e., an event pair of the inverse of complementary ESG, is defined as a faulty event sequence (FES). An FES is complete (FCES=faulty complete event sequence) if the sequence starts from the entry node. The union of the sets of FCESs of the minimal total length to cover the FESs of a required length is called *Minimal Spanning Set of Faulty Complete Event Sequences* (MSFCES).

In comparison to the interpretation of the CESs as legal walks, illegal walks are realized by FCESs that never reach the exit. An illegal walk is minimal if its starter cannot be shortened. Assuming that an ESG has *n* nodes and *d* arcs as EPs to generate the CESs, then at most $u{:=}n^2{-}d$ FCESs of minimal length, i.e., of length 2, are available. Accordingly, the maximal length of an

FCES can be n; those are subsequences of CESs without their last event that will be replaced by an FEP. Therefore, the number of FCESs is precisely determined by the number of FEPs. FEPs that represent FCES are of constant length 2; thus, they also cannot be shortened. It remains to be noticed that only the starters of the remaining FEPs can be minimized, e.g., using the algorithm given in [48].

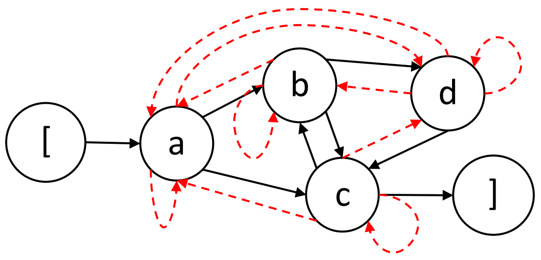


Figure 10. Completion ESG of Figure 5 (a) to determine MSFCES

The minimal set of the illegal walks (MSFCES) for the ESG in Figure 10: *aa, ad, abb, aba, aca, acc, acd, abdb, abdd, abda*

3.2 Exploiting the Structural Features of SUT for Further Reduction of Test Effort

The approach has been applied to the testing and analysis of the GUIs of different kind of systems, leading to a considerable amount of practical experience. A great deal of test effort could be saved considering the structural features of the SUT. Thus, there is further potential for the reduction of the cost of the test process [52].

Analysis of the structure of the GUIs delivers the following features:

- Windows of commercial systems are nowadays mostly hierarchically structured, i.e., the root window invokes children windows that can invoke further (grand) children, etc.
- Some children windows can exist simultaneously with their siblings and parents; they will be called *modeless* (or *non-modal*) windows. Other children, however, must "die", i.e., close, in order to resume their parents (*modal* windows).

Figure 11 represents these window types as a "family tree". In this tree, a unidirectional edge indicates a modal parent-child relationship. A bidirectional edge indicates a modeless one.

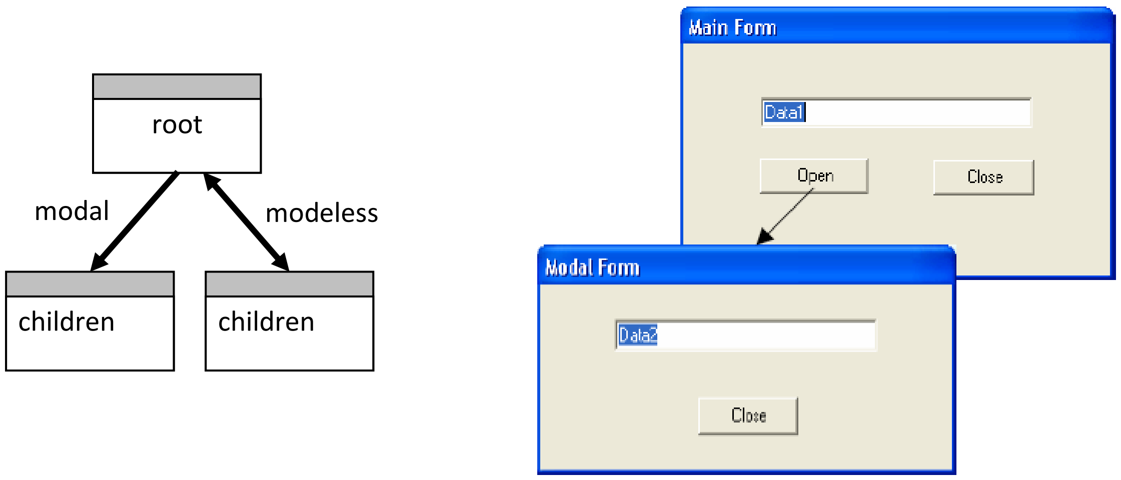


Figure 11. Modal windows vs. modeless windows and an example of a modal opened window

Because modal windows must be closed before any other window can be invoked, it is not necessary to consider the FESs of the parent and children. This is true only for the FCESs and MSFCES as test inputs considering the structure information might impact the structure of the ESG, but not the number of the CESs and MSCESs as test inputs.

Thus, similar to the strong-connectedness and symmetrical features [49], the modality feature is extremely important for testing since it avoids unnecessary test efforts.

3.3 **Case Studies and Their Empirical Evaluation for the Practice**

The objective of the case studies in this section is to determine the increased test effort that arises in relation to the length/number of ESs to be covered and to find out whether this additional test effort is rewarded adequately by the revelation of additional errors. The data needed for this analysis were collected and evaluated by means of experiments carried out in accordance with the principles of software experimentation [50]. Case study 1 focuses on the test effectiveness detecting defects by the various coverage type with different length [53]. The second case study, case study 2, is focusing on the test efficiency, i.e., the test cost reduction achieved from minimal test sequences [45].

3.3.1 Case Study 1: Software Application GUI under Test

For the case study, RealJukebox (RJB) has been selected, more precisely the basic, English version of the RJB 2 (Build: 1.0.2.340) of RealNetworks. There are several reasons why RJB has been selected to be SUT. First, RJB as SUT is a commercial, popular application that is widely well-known and accepted by a great variety of users. Second, the selected SUT has been be used over many years in different languages and in cultural contexts. Furthermore, RJB has been frequently updated and therefore, is mature and well established. Last but not least, RJB makes comfortable use of dynamic window components in several hierarchy levels. The basic configuration of the tested RJB consists of about 200 distinct components. To sum up, choosing the RJB as SUT avoided studying an "alpha" version of a no-name product for the case study with the present approach.

3.3.1.1 Results and Analysis

Table 1 arbitrarily extracts some of the detected faults. The fault reproduction process is very simple. As an example, in order to reproduce the fault No. 1, one starts with the Control option of the Main Menu of the RJB (see Figure 12) and sub-sequentially pushes the Rec button and then FF button. In Figure 13, the dashed line with the label No. 1 uniquely identifies this sequence of actions. The other faults of Table 1can be reproduced the same way.

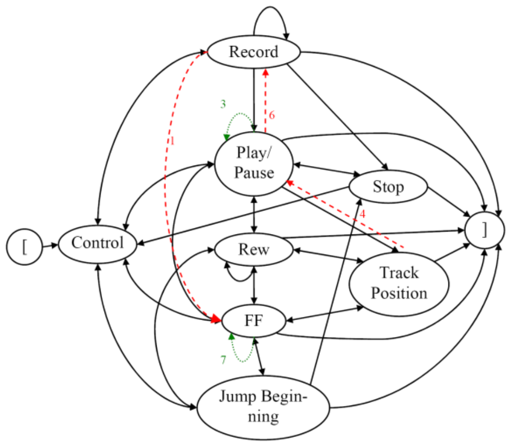


Figure 12. FEP revealed faults in a sub-graph representing "Playing track of Figure 7(*P*)

Table 1. Excerpt from the list of faults revealed by testing the system function "Play and Record a CD or Track"

No.	Detected Faults	Test Case
1	While recording, pushing the forward button or rewind button stops the recording process without a due warning.	*Record FF*
2	If a track is selected but the pointer refers to another track, pushing the play button invokes playing the selected track; i.e., the situation is ambiguous.	*SelectTrack Play*
3	Menu item Play/Pause does not lead to the same effect as the control buttons that will be sequentially displayed and pushed via the main window. Therefore, pushing play on the control panel while the track is playing stops the playing.	*Play Play*
4	Track position could not be set before starting the play of the file.	*Trackposition Play*
5	Record Shuffle does not activate shuffling, i.e., tracks will be processed sequentially.	*CheckOne++ Shuffle Record*
6	If the track is in Pause and Record button is pushed, then the track will be played.	*Play/Pause Play/Pause Record*
7	The system jumps to a track that was not selected and terminates the playback although the selected tracks have not been completely played.	*Play/Pause FF FF FF*

Note that the faults No. 2 and 5 are not included in Figure 12 as they are detected via other ESGs. Due to lack of space, these completed ESGs are not included in this chapter.

Table 2 summarizes the number of test cases, their length and the corresponding faults. Faults are further classified as surprises and defects. *Defects* are serious departures from specified behavior; *surprises* are user-recognized departures from expected behavior. A surprise behavior is not explicitly indicated in the specification of the UI; it should, however, be perceived by some users as a disturbing or disappointing behavior of the system.

Table 2. Test case costs and detected faults depending on the event length

Length of covered ES	Test Cases	Detected Faults by *CES*	Detected Faults by *FCES*	Total nb. Detected Faults	Surprises	Defects	Fault per Test Case (Efficiency)
2	914	24	20	44	16	28	$4.8 \cdot 10^{-2}$
3	2458	24+7	20+5	56	16+8	28+4	$2.3 \cdot 10^{-2}$
4	6936	31+4	25+8	68	24+10	32+2	$0.9 \cdot 10^{-2}$

The number of defects detected by test cases of length 3 and 4 increases obviously slower in relation to those of length 2. Since the faults are independent, these longer tests should still be executed, if the test budget and time allow for this. Another reason why test cases of length 3 and 4 should be executed is given by the likely severity of the “expensive” faults, i.e., defects that can only be detected with these longer, thus more “expensive”, tests. This situation is simple to explain: The longer the test procedure lasts, the less populated the remaining faults become, while one might expect to detect more intricate and subtle faults.

Figure 13 depicts the results found. It can be observed that the tests based on the CESs of length 4 and on FCESs of length 4 are very beneficial in detecting defects: 19 defects have been detected by tests based on FCES of length 4 in relation to only 6 based on FCESs of length 2! Thus, a clear tendency can be observed that an increasing number and length of CES-based and

FCES-based test cases lead to the detection of an increasing number of defects. Note, however, that Figure 13 does not consider the number of necessary tests, i.e., test costs.

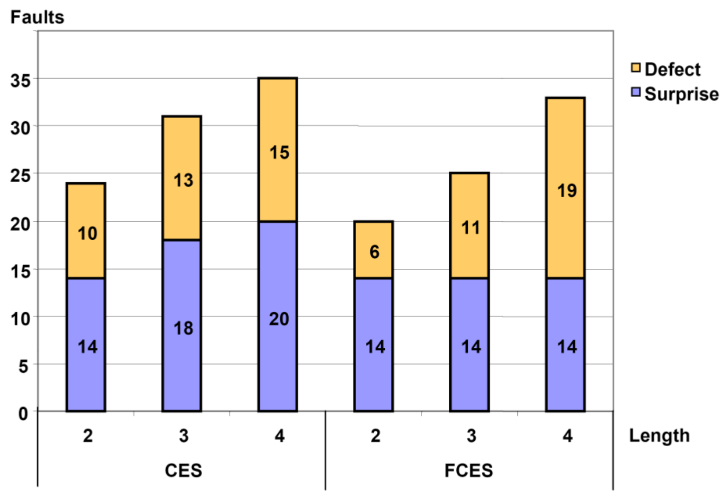


Figure 13. Defects and surprises based on *CES*/*FCES*, depending on the event length

3.3.1.2 Lessons Learned

Now, summarizing the observations of the test process when performing the case study and their implications lead to following major findings:

- The RJB that has been tested in this section is a product that has been matured in many years of intensive and extensive deployment.
- The fact that so many faults could be detected in this product motivates the refinement and improvement of this approach.

To sum up further results, Table 2 and Figure 13clearly show the fact that CES-based tests of length 2 are the most cost-effective ones in this approach, i.e., they detect faults at the lowest costs per fault. In other words, there is a rapid fall off in cost-effectiveness of length of the event sequences to be covered as a consequence of the rapid rise of the number of the test cases (a total of 914 tests to cover ESs of length 2 for all 12 functions whereas 6936 tests to cover their ESs of length 4, cf. Table 2). This finding can be explained by analyzing the structure of the SUT: The number of CESs to cover ESs that are longer than 2 primarily increases with the number of loops within the ESGs. In other words, the more vertices of the ESG under consideration are connected with each other, the larger is the number of tests to cover ESs that are longer than 2.

On the other hand, tests which cover ESs of length 3 and 4 seem to detect more and intrinsic faults (if any), however at considerably more cost per detected fault. Finally, CES-based tests are more cost-efficient than the ones which are FCES-based.

Based on these observations, it is strongly recommended starting the test process with the CES-based test cases of length 2, further continuing with the CES-based test cases of length 3 and 4 and finally to start with the FCES-based ones. If the cumulative number of the detected faults grows very slowly, one can terminate the test process after execution of the CES-/FCES-based test cases to cover the ESs of length 4 as further tests become very cost-ineffective. This really cannot be considered as a tendency of “reliability growth” because the detected faults were not corrected; they have just been ignored; however, the multiple counting of the same faults has been avoided.

The approach delivers a very simple, but nevertheless a cost-effective, stepwise and straightforward test strategy, because the approach enables the enumeration of the test cases and, consequently, the scalability of the test process.

3.3.2 Case Study 2: Embedded Software GUI under Test

The SUT we use in the examples is a control terminal of a marginal strip mower (Figure 14) which controls a marginal strip mower (RSM13) of a special, heavy-duty vehicle (Unimog of Mercedes-Benz). This display unit takes the optimum advantage of mowing around guide poles, road signs and trees, etc. Operation is effected either by the power hydraulic of a light truck, or by the front power take-off. Further buttons on the control desk (Figure 14) simplify the operation, so that, e.g., the mow head returns to working position or to transport position when a button is pressed.

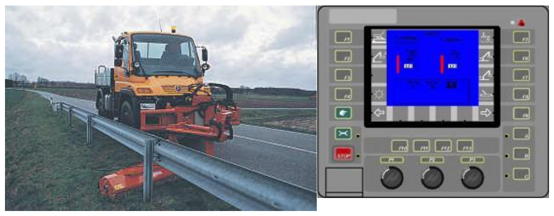


Figure 14. The example vehicle and its display unit as a control desk

3.3.2.1 Results and Analysis

For a comprehensive testing, several strategies have been developed with varying characteristics of the test inputs, i.e., stepwise and scalable increasing and/or changing the length and number of the test sequences, and the type of the test sequences, i.e., CES- and FCESs-based, and their combinations. Following could be observed: The test cases of the length 4 were more effective in revealing dynamic, intricate faults than the test cases of the lengths 2 and 3. Even though more expensive to be constructed and exercised, they are more efficient in terms of costs per detected fault. Further on the CES-based test cases as well as the FCES-based cases were effective in detecting faults.

Due to the lack of space, the experiences with the approach are here very briefly summarized. This can be, however, found in [53]. To sum up the test process, one student tester carried out 826 tests semi-automatically and detected a total of 39 faults, including some severe ones (Table 3).

Table 3. Two of the detected faults of the RSM control terminal

No.	Faults Detected by the FCES
1.	The cutting unit can be activated without having any pressure on the bottom, which is very dangerous if pedestrians approach the working area.
2.	Keeping the button for shifting the mow head pushed and changing to another screen causes control problems of shifting: The mower head with the cutting unit cannot immediately be stopped in an emergency case.

In a second stage, the results of the research work for minimizing the spanning set of the test cases (MSCES and MSFCES) have been applied to the testing of the margin strip mower. Table 4 demonstrates that the minimization algorithm (Section 3.1.1) could save in average about 65 % of the total test costs, while the exploitation of the structural information (Section 3.2) of the SUT could further save up to almost 30 %.

Table 4. Reducing the number of test cases

Length	#CES	# MSCES	Cost Reduction ES
2	40	15	62.5 %
3	183	62	66.1 %
4	549	181	67.0 %
Sum	772	258	65.2 %

This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. **Citation information**: Fevzi Belli, Mutlu Beyazıt, Christof J. Budnik and Tugkan Tuglular, Advances in Model-Based Testing of Graphical User Interfaces. In: Atif M. Memon, editor, *Advances in Computers, Vol. 107*, Burlington: Academic Press, 2017, pp. 219-280., ISBN: 978-0-12-812228-0

Length	# MSFCES without structural information	# MSFCES with structural information	Cost Reduction MSFCES
2	75	58	22.7 %
3	167	218	35.7 %
4	487	292	40.0 %
Sum	729	568	32.8 %

3.3.2.2 Lessons Learned

Lesson 1. Start Small, but as Early as Possible

The determination and specification of the CESs and FCESs should ideally be carried out during the definition of the user requirements, much before the system is implemented; the availability of a prototype would be helpful in this task. They are then a part of the system and the test specification. However, CESs and FCESs can also be produced incrementally at a later time, even during the test stage, in order to discipline the test process.

As a strategy, one starts with the CESs and FCESs that cover all event pairs. Test results and quality targets determine how to proceed further, i.e., whether to consider testing with event triples and quadruples.

Lesson 2. Good Exception Handling is not Necessarily Expensive but Rare

Most GUIs subjected to tests do not consider the handling of the faulty events. They have only a rudimentary, if any, exception handling mechanism, realized by a “panic mode” that mostly leads to a crash, or ignores the faulty events. The number of the exceptions that should be handled systematically, but have not been considered at all by the GUIs of the commercial systems is presumed to be on an average about 80%.

Lesson 3. Analysis Prior to Testing Can Reveal Conceptual Flaws

The analysis of ESGs of the GUIs of some commercial systems has revealed several conceptual flaws: absence of edges, indicating incomplete exception handling, and missing vertices or events (approximately 20%). This amounts to defective components in the final product, highlighting the flaws in the initial concept and the process of product development. In this connection, the proposed approach offers an important unexpected benefit: it provides a framework for the accelerated maturation of the product and for exercising the creativity of the developers.

4 Contract-Based Testing of GUIs

Methods and techniques presented in Section 2 and Section 3 aim to generate abstract test cases. In this Section, a contract-based approach for concrete test case generation is presented. The presented contract-based approach stems from the concept of design-by-contract (DbC). Meyer [54] introduced DbC as an object-oriented design technique. Design by contract follows the principle that interfaces between modules of a software system should be governed by precise specifications, similar to contracts between humans or companies. The contracts cover mutual obligations (preconditions), benefits (postconditions), and consistency constraints (invariants) [55]. From the point of view of GUI testing, DbC plays an important role because contracts delineate what user is expected to provide as input and what the GUI is expected to supply as output with respect to the provided input. From testing point of view, a GUI operation can be evaluated with respect to preconditions, postconditions, and invariants according to DbC. Following DbC, UML is supplemented with object constraint language (OCL) to provide some formalism. Contracts in decision table format are compact and easy to understand and maintain [56]. However, there is no generally accepted formalism for contract representation.

4.1 Contract-Based Testing in General

Contracts form a valuable source of information regarding the intended semantics of the software. A behavioral specification is a description of what is expected to happen when software executes [57]. This specification can be used to verify that the software meets its requirements. When behavioral specification is presented using DbC, it becomes more useful for both programmers and testers.

There exist some approaches that adopt the DbC-idea for testing. Zheng and Bundell [58] introduced an UML-based software component testing technique called Test by Contract. Ciupa and Leitner [59] noted that the validity of a software element can be ascertained by checking the software with respect to its contracts. In addition to using contracts to automatically generate test input values, contracts can be used as test oracles as they define valid and invalid conditions for the software. Thus, utilization of contracts eliminates the necessity of developing a test oracle for each test case [60]. As contracts are used to evaluate test results, the quality of the test oracles is entirely dependent on the quality of the contracts [59]. Aichering [61] shows how mutation testing can be applied to contracts. Similar to source code mutation, a mutant contract is produced by introducing small change to the formal contract definition. Then test-cases that are able to detect the introduced mutations are selected.

Madsen [62] investigated how JUnit and Design by Contract can be combined. This way, assertions written as pre- and post-conditions and class-invariants can be used in unit test cases and automatic execution and evaluation of test cases becomes possible. If test cases are automatically generated, then JUnit can be used to setup and execute the test cases and then contracts can be used to evaluate the test cases. Languages like Phyton, C++, Java are extended to comply with DbC for catching bugs. Guerreiro [63] used design by contract in C++ by using and inheriting the Assertions class. In [64], the DbC concept is integrated into the programming language Python and adopted by adding mechanisms for dynamic type checking of method parameters and instance variables.

Contracts establish the ground for the automation of the model-based testing process. While testing a system, a model of the system helps to predict and control its behavior. Modeling a system acquires the understanding of its abstraction, and there is the need of a formal specification technique for distinguishing between legal and illegal situations. Contracts serve perfectly for these purposes. There are some contract-based testing techniques focused on web service testing [65],[66],[67], where web service behavior is modeled using contracts. Valentini et al. [68] proposed a framework for contract-based component testing, which enables extendable and robust contract-checkers to be dynamically inserted between client component and supplier component. Contract-checkers use the contract between client and supplier to work like proxies by forwarding method call to the client, the result back to the supplier and to evaluate test result. Xu et al. [69] proposed an approach that transforms a contract-based test model into an operational model, which enables analysis of the correctness of the test model. Then integration tests are generated to meet coverage criteria of the test model.

Another use of contracts is for robustness. Robustness is a quality attribute, which is defined by the IEEE standard glossary of software engineering terminology [70] as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. Specifications presented in contracts helps to improve robustness of software [71]. An example of contract usage in robustness testing is presented by Tuglular et a. [72], where they introduced decision table augmented ESGs, which utilizes design by contract patterns, and applied this concept to event-based robustness testing for catching boundary overflow errors.

4.2 ESG-Based Contract Testing of GUIs

The contract notion is used to describe input properties in precise terms. Preventing invalid input from ever getting to the application in the first place is possible only at the user interface. Therefore, GUIs should be specifically designed to filter unwanted or unexpected input. This can be achieved through input contracts that are defined and used in our work. Model-based specification of input contracts is achieved through an input contract model, which enables the input data and corresponding actions to be defined with their constraints. Thus, for simplicity, the term “testing" here is used to refer to function-based, specification-oriented testing, or black-box testing.

In the input contract testing approach, the tests are derived from contracts supporting the creation of test input values and test oracles. This novel approach suggests that an automatic input testing process is possible with a GUI test driver that invokes mouse clicks and enters text into rich client GUIs. In this context, contracts form a valuable source of information regarding the intended semantics of the software. As noted by Ciupa and Leitner [59], the validity of a software element can be ascertained by checking the software with respect to its contracts. Therefore, contracts establish the ground for the automation of the testing process. Accordingly, the primary goal of input contract testing is to develop and implement a fully automated test case generation for contract-based GUI input testing.

The input contract testing approach suggests converting graphical user interface specification into a model, which is employed to generate positive and negative test cases. The event sequence graph (ESG) is chosen for the specification of GUIs. ESG merges inputs and events and turns them to vertices of an event transition diagram for easy understanding and checking the behavior of the GUI under consideration.

4.2.1. Input Contract Model

Modeling input data, especially concerning causal dependencies between each other as additional nodes, inflates the ESG model since vertices represent events and edges allowed sequences of events and not transitions as in automata theory. Assuming that a condition for choosing input data can be evaluated to true or false, the combination of conditions results in $2^{|C|}$ combinations, where |C| represents the number of conditions. Each combination of conditions would have to be modeled as vertex and is to be connected with the appropriate successor. Thus a decision table (DT) with n binary conditions subsumes 2^n nodes to realize a thorough evaluation considering all combinations. To avoid this inflation, decision tables are introduced to refine a node of the ESG. Such refined nodes are double-circled. The successors of such refined vertices represent the actions of the DT and vice versa.

A *Decision Table* $DT = (C, A, R)$ represents actions that depend on certain constraints, where

- $C \neq \emptyset$ is the set of *constraints* (*conditions*) as Boolean predicates,
- $A \neq \emptyset$ is the set of *actions*, and
- $R \neq \emptyset$ is the set of *rules*, each of which triggers executable actions depending on a certain combination of constraints.

Decision tables are popular in information processing and are used for testing, e.g., in cause and effect graphs. A decision table (DT) logically links conditions (”if”) with actions (”then”) that are to be triggered, depending on combinations of conditions (”rules”).

Let R be defined as the set of rules, each of which triggers executable actions depending on a certain combination of constraints.. Then, a *rule* $r \in R$ can be defined by

$r = (C_{True}, C_{False}, A_x)$, where

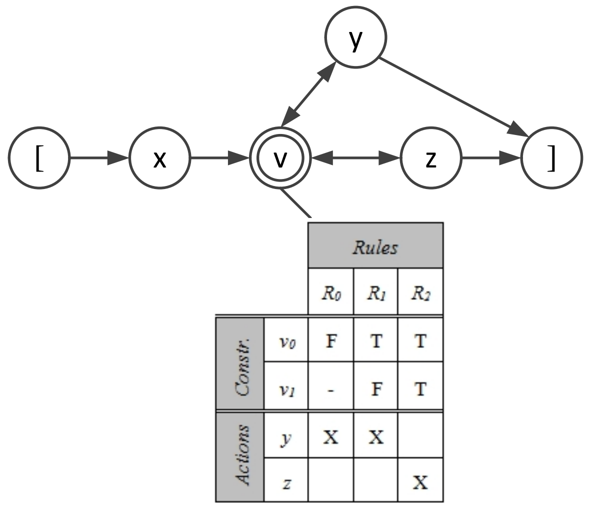
- $C_{True} \subseteq C$ is the set of constraints that have to be resolved to true,
- $C_{False} \subseteq C$ is the set of constraints that have to be resolved to false, and
- $A_x \subseteq A_{ui} \times A_{xcpt}$ is the set of actions that should be executable if all constraints $t \in C_{True}$ are resolved to true and all constraints $f \in C_{False}$ are resolved to false with
  - A_{ui} containing possible user interactions,

- A_{xcpt} containing exception messages.

That is, one rule represents a specific combination of conditions where each condition is evaluated either to true or to false. Depending on one rule, one or several follow-on actions are allowed. In the other way around, the execution of a specific action is only allowed if input data is chosen along a rule which possesses the considered action as allowed successor. As already stated above, the combination of conditions results in $2^{|C|}$ combinations, that is, $2^{|C|}$ rules can be formulated without producing redundancy. Note that $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \emptyset$ under regular circumstances. In certain cases it is inevitable to mark conditions with a *don't care* (symbolized with a '-' in DT), i.e., such a condition is not considered in a rule and $C_{True} \cup C_{False} \subset C$. A DT is used to refine data input of GUI's.

An example of DT is given Figure 15. This DT can be used to refine a node of an ESG. This node will be double-circled and next event, which is an action in the DT, is decided with respect to DT that is attached to this double-circuled node. Such an ESG is called DT-supplemented ESG and is shown in Figure 15.

For DTs, such as the one presented in Figure 15, X entry indicates an action, or for GUIs a user interaction. No exception is defined for actions *y* and *z*. As an example, rule 1 (R_1) reads as follows: **If** v_0 is resolved to *true* and v_1 is resolved to *false*, **then** action *y* will be executed. If this DT is used to refine a node of ESG, such as given in Figure 15, then regarding to R_1 next event after *v* will be *y* and the ES will be (…,*v*,*y*, …).



		Rules		
		R₀	*R₁*	*R₂*
Constr.	*v₀*	F	T	T
	v₁	-	F	T
Actions	*y*	X	X	
	z			X

Figure 15. An example of DT-supplemented ESG [73]

Given a GUI, the quadruple σ is an *input contract model* (*Io, Dv, Ac, Co*), where

- *Io* is the finite set of *GUI input objects*;
- *Dv* is the finite set of *data variables*;
- *Ac* is the finite set of *actions on GUI*;

- *Co* is the *input contract definition* represented by a *DT.*

As stated before, event sequence of GUI is modeled with DT-supplemented ESG. It is an ESG with a special DT, where conditions of DT come from constraints of input contracts. Input contract model provides a guideline for the construction of a contract-supplemented ESG for the GUI it represents. Input objects, such as inputArea and comboBox, and button objects indicate possible events. Event sequences are established among these events through drawing edges between vertices.

4.2.2 Input Contract Testing Process

For GUI input contract testing, test scope is always a GUI. A set of GUI components that make up a window can be tested if event-based testing is integrated into input contract testing. Therefore, GUI input contracts are modeled with contract-supplemented ESGs so that a seamless testing process can be achieved for a window or a composition of GUI input elements. Solutions for automating test case generation and test result interpretation stages are described in the following paragraphs.

A *test case* specifies input values for a method of an input component, which may work on one or more input area. A *test suite* is composed of test cases to check validation of all assertions offered by an input contract. The input values making up a test case can be derived from the constraints of a provided contract. Expected outputs are actions with or without exceptions given in DT. Please note that an input contract is not supposed to cover all inputs, its purpose is to filter.

GUI input contract testing process is given in Algorithm 7. Full event coverage and full rule coverage criterion is fulfilled in terms of coverage. For full event coverage criterion, each event is executed at least once. In other words, each node of ESG is visited at least once. For full rule coverage criterion, each rule should be tested independently. These test cases should be sampled from input space composed of valid and invalid values of constraints.

Algorithm 7. GUI input contract testing process

```
generate the corresponding ESG
cover all events by means of CESs
foreach CES with decision tables do
   generate data-expanded CES using corresponding DT (input contract-
       based test case generation)
apply the test suite to GUI
observe GUI output to determine whether a correct response or a faulty
   event occurs
```

4.2.3 Input Contract Test Generation

The DT is used to produce test cases automatically. Since it is often not feasible to include all possible input values for a test case, the central question of testing is about the selection of test input values most likely to reveal faults. This problem comes down to grouping data into equivalence classes, which should comply with the property that if one value in the set causes a failure, then all other values in the set will cause the same failure. Conversely, if a value in the set does not cause a failure, then none of the others should cause a failure. This property allows using only one value from each equivalence class as a representative for its set.

Equivalence class testing divides the test value domain into equivalence classes using contract conditions. Each test case selects one input value from each equivalence class. This approach is improved by boundary value selection of input values for numeric and date data, which appear at the boundaries of equivalence classes. For string data, such as names, and for other types of data, such as files, a set of input values representing each equivalence class should be manually prepared in advance with respect to the input contract and then test input values are selected randomly for each equivalence class. Thus, in our work, cause-effect testing, which generates test values from decision tables, is used to strengthen equivalence class testing. In the presented approach, causes are input conditions and effects are represented by actions. This proposed approach is presented in Algorithm 8, namely input contract-based test case generation algorithm, which derives test inputs from contract-supplemented ESG.

The input contract-based test case generation algorithm produces test values for each rule in the DT. The DT is represented with a data structure that contains the set of variables, the set of clauses along with its variable(s) and their equivalence classes, the set of actions and exceptions, and the set of rules wherein each rule is composed of a conjunction of clauses and conjunction of actions and exceptions.

For each rule, the function findTestInputValue is called. It attempts to find values for variables that satisfy the Boolean expression that is a special case of a *constraint satisfaction problem* [74] of the corresponding rule in the DT. The function solveCSP determines valid and invalid equivalence classes for each clause and searches the values that make the Boolean expression true. The runtime complexity of the whole algorithm mainly depends on this function, which has to be solved for each rule of the DT.

The algorithm of getAssignment within the function solveCSP starts by assigning a value to a single variable and extends the solution step-by-step with the other variables by assigning values. If a value assignment to the current variable is not possible due to previously selected values, the algorithm steps back and chooses next value from the set of boundary values for the current variable. This procedure is also called "simple backtracking". The proposed algorithm combines backtracking with the techniques "Arc Consistency Check" and "Minimum Remaining Values", see [74] for further information, to solve the given constraint satisfaction problem modeled by DT.

Algorithm 8. The input contract-based test case generation algorithm.

```
foreach event with DT do
    foreach rule in DT do
        findTestInputValue(DT, rulei)
function findTestInputValue
begin
  tc_inputs : Test_Case_Inputs
  set_clauses : LIST[<Clause, Variable, EquivalenceClasses]
  b : BooleanExpr
  set_clauses ← getClauses(DT)
  b ← determineBooleanExpr(DT, rule)
  tc_inputs ← solveCSP(b, set_clauses)
  return tc_inputs
end
function solveCSP
begin
  assignment : LIST[<Variable , SelectedValue>]
  g ← getConstraintGraph(b, set_clauses)
  assignment ← getAssignment(g, assignment)
  return assignment
end
```

The runtime complexity for backtracking is given as O(n*d) where n is the number of nodes for the corresponding constraint graph and d is the depth of the graph. The runtime complexity for the consistency check is given as O(n^2d^3) [74]. However, in practice the number of variables on a GUI is strictly limited due to usability restrictions.

Simultaneously, this also limits the number of corresponding constraints so that the runtime complexity of this algorithm is negligible. Furthermore, the search space for numerical values may be narrowed by considering only boundary values of equivalence classes. Finally, the function solveCSP returns test case inputs for a rule in the decision table. Resulting test cases contain test input values as well as expected results.

The development of test oracles, which automatically performs a pass/fail evaluation of the test case, is an important issue in software testing. Developing such test oracles manually when writing test drivers is expensive and error-prone. Since our work proposes that assertions based on contracts can effectively be utilized as test oracles, the presented methodology is composed using different techniques to derive the oracles from the contracts in synchronization with the generation of test input values.

Fully automated testing requires automating the handling of oracles. In this case, evaluation of test results is straight forward due to the presence of contracts as specifications. Test cases are generated with expected test results automatically from the DT, which is constructed from input contracts. Since the test oracle in this approach uses executable input contracts by means of checking test case results, test outputs can be easily compared with expected test results. Thus, the test oracle in our work enables an automatic pass/fail evaluation of the test case. If the obtained results match the expected results, then the test case passes, otherwise it fails.

Analysis of the test case generation process reveals the fact that ESGs are to be transformed into one large model for test case generation. On the other hand, DTs could be consolidated, which results in reduced number of rules. Both facts give some clues about the scalability of the presented approach. Transforming ESGs into one large model might complicate test case generation and the intuitive partitioning of SUT intended by the tester would be lost. Further test generation techniques are considerable, which make use of the intuitive partitioning of the tester to reduce and/or simplify test sequences and their generation, especially with regard to input contracts. The more input contracts exist, the costlier is their evaluation. This is due to the fact that adding just one single input contract doubles (in the worst case) the number of combinations of input contracts to be tested. Thus, further techniques to reduce the evaluation complexity of large sets of input contracts could be helpful, such as partitioning of input contracts that could be achieved by a hierarchical set of DTs.

The following questions are required to be answered in the future: a) How much overhead does the presented approach impose on a tester of large software systems? b) How far can a tester develop contracts for such an application and how long would it take? c) How would a tester developing contracts for applications impact speed of execution during testing? Moreover, developing formal semantics for input contracts will have an important impact not only on the testing of GUIs but also on the design and implementation of GUIs. Definition of refinement and inclusion operations on contracts provide distinct means to express complex input behavior in terms of simpler behavior. Furthermore, a refinement enables specialization of contractual obligations and invariants of other contracts, whereas inclusion allows contracts to be composed from simpler sub-contracts [75].

5 Rationalization and Automation of GUI Testing

Techniques for modeling, analyzing, and testing GUIs are represented in the previous sections. This section categories and exemplifies tools for GUI testing that are available on the market. It is evident that this market changes frequently as these tools are to a great extent short-living. So, the critical reader may forgive if the authors have forgotten to name some important brands.

5.1 MBT Test Tools in General

MBT is defined as an approach to deriving executable tests from a given model of the system under test (SUT) using several test selection criteria. The model is built either from requirements and/or from specifications of test model. The model should contain both input and expected output in order to be able to generate test oracles [83]. Thus, no model-based input generator and no test automation framework where test cases can be manually created or pre-recorded [76] are considered to be MBT tools.

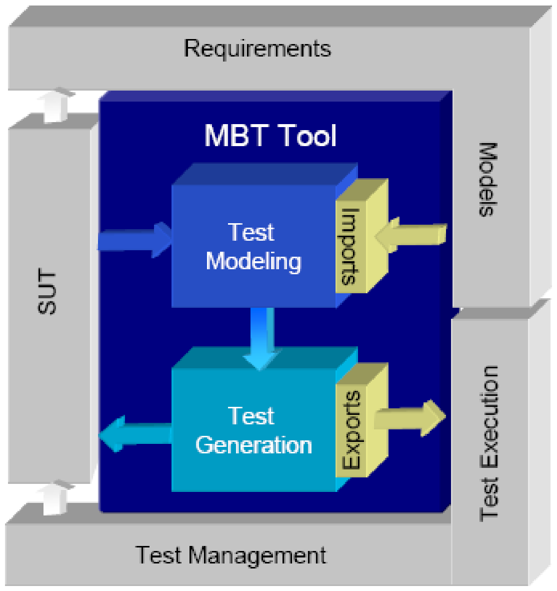


Figure 16: MBTT Categories of Interest

An MBT tool supports the test life cycle and interacts with other development phase elements as depicted in Figure 16. Selected evaluation criteria are (i) Test Modeling - design of the test model derived from the SUT, (ii) Test Generation – strategy for deriving test cases, and (iii) Extensibility – integration possibilities with other tools via import/export interfaces and/or extending tool to different domains. The usability criterion has been intentionally omitted. Test generation algorithm are better covered elsewhere and omitted here.

MBT tools need to satisfy the following criteria: (i) Usage of a test model (not a system model) from where tests are derived. (ii) Automate test generation covering test input data and system behavior. (iii) The test model is represented by a formal modeling language.

5.2 Test Tools for Graphical User Interfaces

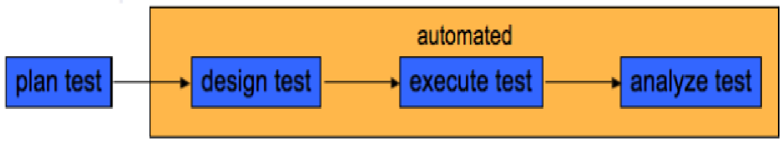
The term test automation for graphical user interfaces (GUIs) in industry implies automated test execution in most cases. There are many GUI test tools available [77], [78] which we refer to as a selection of commercial as well as open-source tools. The test tool market however is changing rapidly. This makes it very difficult to discuss or compare specific tools which will be sustainable at the market. Thus, this section rather focuses on the needs from a practical and industrial perspective.

Test execution itself is mostly automated by *Capture/Replay* tools such qf-test [84] or jfcUnit [85] and is used for regression testing. More recently, *Capture/Replay* tools have been enhanced to visual GUI test tool using image recognition instead of GUI object code or coordinates [90]. Test automation is more than just regression testing. Automation should be attempted for as many stages of the entire test process as possible. First, the test process has to be managed, which means that the test documentation, the different releases of test cases, and the test cases itself have to be maintained and kept consistent with the requirements they are associated with. In practice, this is the most neglected step. In the next step, the test cases have to be generated. This differs from a simple test script generator, which only allows defining test inputs and their corresponding outputs. When a specification represented as a model is used to automatically generate executable test, the method is referred to as *model-based* test generation. In contrast, in *data-driven* test generation [86], existing test cases of just plain templates are parameterized with different data the test has to be run with. Once the test cases are designed, tests have to be conducted on the System under test (SUT). Automation of this test step is supported by a wide variety of tools such as Conformiq Test Generator [91] and Smartesting CertifyIt [92] that support many popular programming languages. Finally, the test analysis step is left in which conducted tests and their outputs are evaluated. The results of the analysis are needed to fix the faults and to decide further tests.

The test automation has to keep through to the complete product life cycle. D. Kelly [79] divided the life cycle into six major stages and for each stage there is a checklist of questions. Depending on the responses it is decided whether to automate the corresponding stage or not. However, the work [79] do not include necessary detail how to set up test automation during the entire life cycle. Another work dealing with the issue about when a test should be automated is given in [80]. It assumes the intent of automated testing and does not take a decision on need for automation. Beyond this it is still difficult to decide which tool fits best to the requirements. This is addressed in [81] providing a catalogue of features to be compared when evaluating a GUI test tool.

5.3 Test Automation in Theory and Practice

This section describes test automation of the entire test process, divided into four parts as illustrated in Figure 17, in the context of black-box testing of software applications containing a graphical user interface (GUI). The objective of test automation research is to maximize automation in test process. This means that existing models from development are used to generate test cases (model-driven) using suitable algorithms. The models have to contain the test inputs and the outputs to derive a fault model. This ensures that the tests are observable and the outputs can be compared with the expected ones described in the model. It may be observed in Figure 17 that the test process steps are coordinated, i.e., the test results of one stage can be used as an input of the following stage. Therefore, the entire test process is automated as a single step.



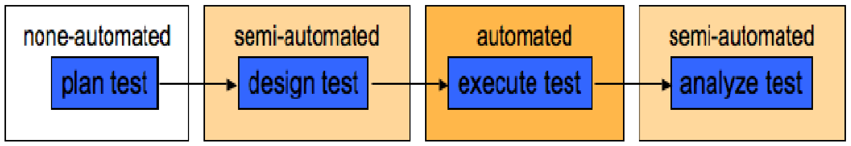


Figure 17. Gap between Test Automation in Theory and Practice

Insights from industrial projects trying to adopt test automation reveal that test automation is treated for each step separately. This differs from theory which treats test automation as a single step. In Figure 17, steps which still need some manual work are depicted as *semi-automated*. The test planning step is done manually and is accordingly depicted as *non-automated*. In addition, automation is adopted in most cases too early which means without any preparation [80].

The most reasonable cause to implement test automation in practice is for repeating test cases. That is also the reason why until now regression testing is so widely used and test automation is almost limited to the test execution in industry. Regression testing just confirms that despite changes to the software, the tests previously run provide identical results now. In fact, this means that no new functions have been tested. So regression testing is not supposed to find new faults in the code. In theory, it is the methodology that reveals fewest bugs. However, the most Capture/Replay tools are not able to provide expected benefits.

5.4 Test Tool Requirements in Industry

For testing GUIs of software systems powerful commercial tools Ranorax [87], Squish [89] and HP Unified Functional Test [88], are available. Nevertheless, numerous time consuming and error-prone manual steps are still necessary to complete the test process. Still, not all test tools allow constructing templates which can be run for different data from a database. Also, not all testing tools nowadays support the use of stubs or wrapping. From the industrial point of view there are many other problems concerning how to implement successful test automation. This section lists practical issues faced while attempting to introduce test automation for graphical user interfaces supported by Capture/Replay tools.

5.4.1 Dynamically Design Changes

Recorded objects of a GUI are identified by its unique properties. But these properties are static. This is an issue as only those objects which are available at the point of recording can be recognized by the tool. Otherwise, the objects have to be introduced into the tool afterwards. But, this can be very time consuming because the system has to set in the right state where the elements are active. Another problem arises due to the changes of different objects from release to release. Although, previous mentioned test tools (Ranorex, Squish, HP UFT) already provide to change the objects properties or to set regular expressions fitting the object, this has to be done manually and takes a lot of time to maintain. A significant problem is the case in which the object will change dynamically depending on the behavior of the user. For example, a conformed GUI depending on the user behavior affected by knowledge based programs or learning software.

There it is impossible to predefine all possible cases where objects will change. These vulnerabilities can be eliminated by a model-based test generation which maintains the model instead of repairing the test cases.

5.4.2 Failure Treatment

Failure treatment is rarely supported by the test tools. One problem observed when testing the GUI is that while the GUI objects are checked, the underlying system operation is not. Indeed, one can write a test scenario which tests if all fields have to be filled out and if the submission button is disabled after sending once. This restriction is enforced to prevent multiple entries. However, this does not prove that the system has really connected with the database and stored or changed the values. Therefore, the test tools have to be extended by a functionality to compare database values with the expected values. Another kind of fault appears when the application crashes. Then a fault can be detected but the bug cannot be fixed, because there is no information on which test cases failed. The only way to find that out is by conducting the test cases manually. This is analogous to finding a zero point numerically. Hence, the advantage of executing the test scripts without being present is lost. The same holds for faults which prevent the process from continuing like open windows. It is assumed that each test case starts at the main window, i.e., from the same starting point. Therefore, the SUT has to be reset in its initial state from which each test case can be started. This is important if the test process should also provide test cases which are expected to fail. Bret Pettichord mentions this special case of resetting the system in his work where he called it an "Error Recovering System" [82]. For bug fixing, it is also recommend having a function which can set the system in predefined system state from which the test can further run. Additionally, it may be necessary to have a memory map which is recorded during the test process. A worst-case scenario occurs when a test case crashes not only the application, but also the testing tool or even the whole system. In this case the only chance to overcome the problem is to start the test execution from a remote machine which has to be provided by the test tool or even to support a test execution on a virtual machine.

5.5 **ESG Test Suite Designer**

Number of tools providing model-based testing is very limited. The ESG tool is a good example of model-based testing tool. The aim of this subsection is to explain ESG tool named ESG Test Suite Designer (ESG-TSD) [93] that provides necessary functions to develop ESG models and to generate test cases form ESG models.

Finite state machines are models used in model-based testing. UML statechart is a way to represent finite state machines. Ticket machine is used as a running example in this subsection. The running example has two ticket machines. Simple ticket machine lets users to buy a single type ticket whereas complex ticket machine allows users to buy tickets by selecting ticket type as well as number of tickets. Their statecharts are given in Figure 18 and Figure 19, respectively.

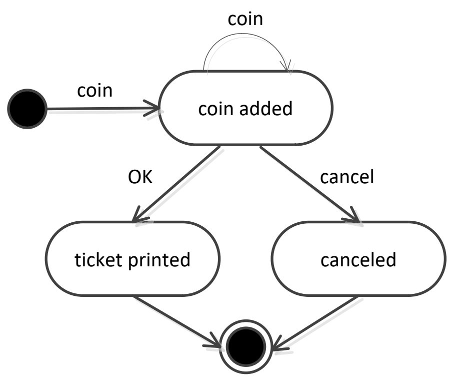


Figure 18. **Statechart for simple ticket machine [94]**

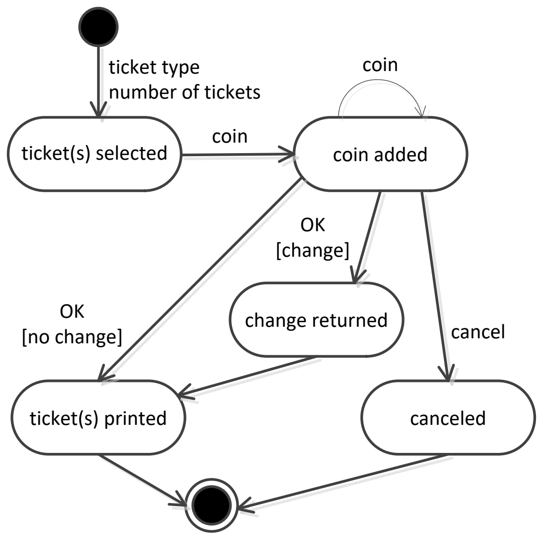


Figure 19. **Statechart for complex ticket machine [95]**

Simple ticket machine shows coins inserted. User either inserts coins until the exact cost of ticket is reached and gets the ticket or quits and gets coins back. Complex ticket machine lets the user to select ticket type and number of tickets and then waits for the amount to be inserted. Once the inserted amount is equal to or more than the required amount then change is returned if there is and ticket(s) are printed. The user can quit while inserting coins.

A statechart can be transformed into an ESG by presenting transitions between events that cause state transitions in statecharts. ESG for simple ticket machine is modeled using ESG-TSD and shown in Figure 20, which shows valid event sequences for simple ticket machine. Complete event sequences, which are given in Figure 21, are generated by pressing wheel icon on the icon tab of ESG-TSD. The output shown in Figure 21 also presents the time elapsed in producing complete event sequences.

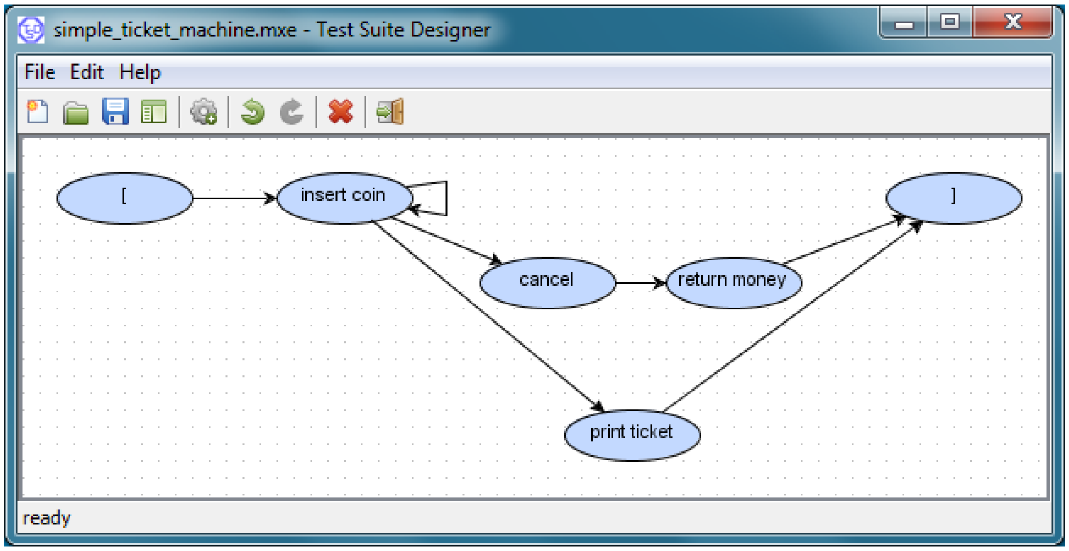
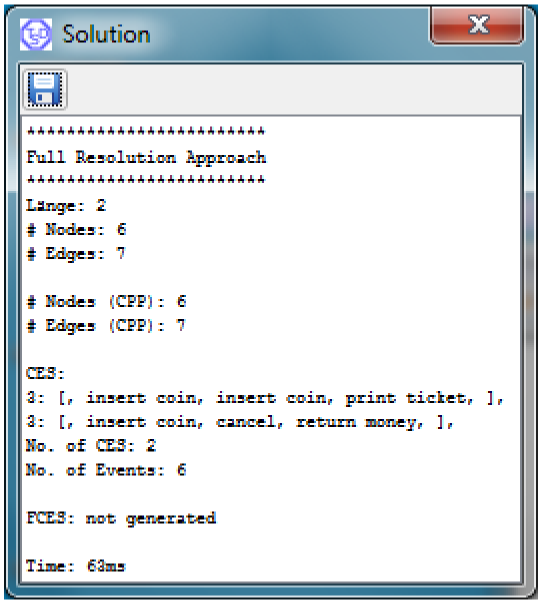


Figure 20. ESG for simple ticket machine in ESG-TSD [96]



```
************************
Full Resolution Approach
************************
Länge: 2
# Nodes: 6
# Edges: 7

# Nodes (CPP): 6
# Edges (CPP): 7

CES:
3: [, insert coin, insert coin, print ticket, ],
3: [, insert coin, cancel, return money, ],
No. of CES: 2
No. of Events: 6

FCES: not generated

Time: 63ms
```

Figure 21. Complete event sequences generated by ESG-TSD for simple ticket machine [96]

ESG for complex ticket machine is shown in Figure 22. When wheel icon is pressed, it generates three complete event sequences as follows:

- "[, select ticket type, enter number of tickets, insert coin, print ticket(s),]"
- "[, select ticket type, enter number of tickets, insert coin, cancel, return money,]"
- "[, select ticket type, enter number of tickets, insert coin, insert coin, print ticket(s), return change,]"

ESG-TSD also produces faulty complete event sequences, which are test cases for negative testing. For instance, "[, select ticket type, insert coin,]" is a faulty complete event sequence of length two for complex ticket machine. ESG-TSD generates 15 faulty complete event sequences for simple ticket machine and 48 faulty complete event sequences for complex ticket machine.

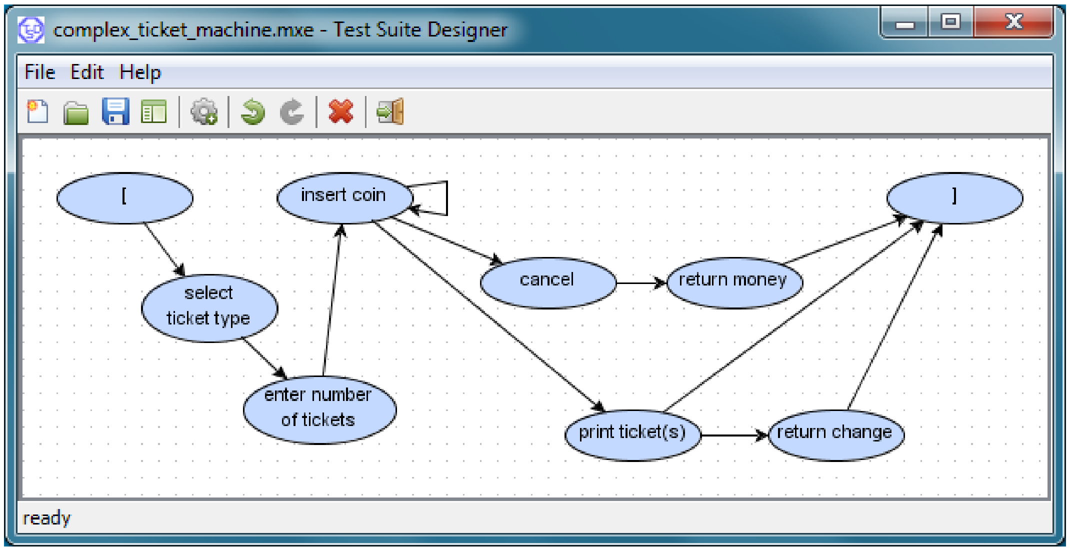


Figure 22. ESG for complex ticket machine [96]

DT-supplemented ESGs are supported by the ESG-TSD. As explained above, DTs help to reduce complexity of ESGs by representing conditional event transitions in DTs within sub-ESGs. DT-supplemented ESG for complex ticket machine is given in Figure 23. The difference between Figure 22 and Figure 23 is that "print ticket(s)" and "return change" events are encapsulated in "tickets & change" sub-ESG represented using double circle. In this sub-ESG after printing ticket(s), if there is no change to be returned, pseudo-exit is executed, whereas if there is a change, after printing ticket(s) first change is returned and then pseudo-exit is executed. This is an example of how to simplify ESG models.

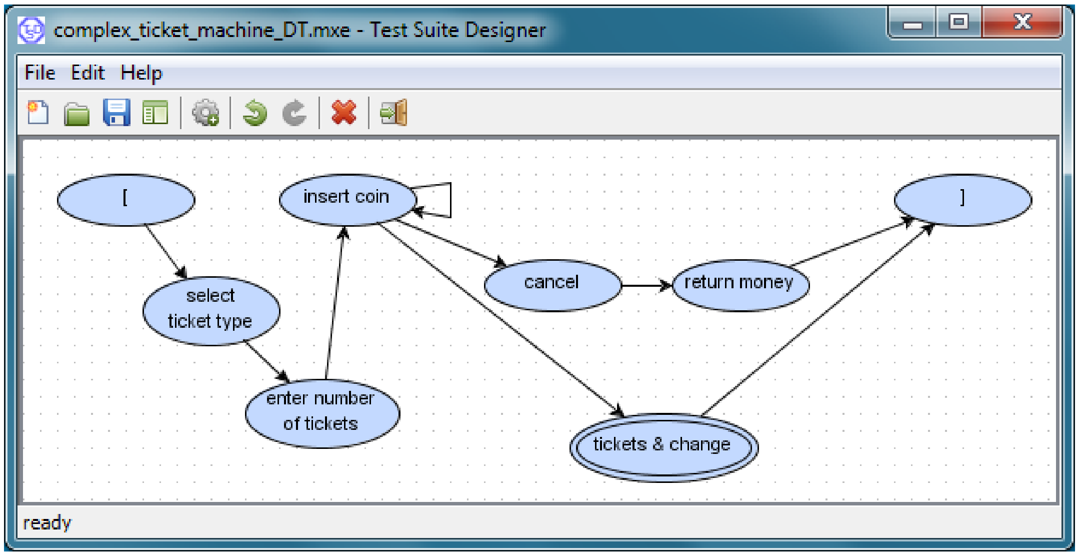


Figure 23. DT- supplemented ESG for complex ticket machine [96]

6 Conclusions

GUIs are likely to continue playing a role in human-computer-interfaces. The question is how far new modes of interaction may make the field obsolete. In other words, can we reuse and generalize some of the things we have learned with GUI testing and apply them to other types of software?

Before wrapping up and summarizing the present chapter, the authors briefly discuss on where they see the field going.

6.1 Peer into the Future

6.1.1 Changes in Look and Feel of UIs will Come, But What About Their Modeling?

GUIs have been important parts of many software applications to provide interactions between the user and the system. Recent advances suggest that UIs are likely to change form in future. Although classical GUI elements such as buttons, menus will still probably be there for at least some time, the way users interact with systems is expected to change greatly. It will not be surprising when interacting with systems using additional types of triggers such as sounds, hand/body/face/eye gestures and brain waves becomes a common thing. The interaction methods supported by UIs have been getting enriched by such technological advances. Therefore, the approaches that do not have right level of abstraction, or that are strongly dependent on certain GUI properties or components are expected to be harder to reuse or adapt in future when the changes reach at a considerable level. However, event-based approaches as discussed in Section 2 can still be useful if the user interact with the system in a way that can be formulated in terms of discrete stimuli. It is always possible that semantic extensions will be required to capture different types of behaviors, descriptions or problems; however, the formal basis will be the same.

On the other hand, the approaches proposed for GUI testing can also be used for other types of software applications with certain adjustments depending on the application area. As mentioned above, if user-system interactions can be formulated in form of events, most of the approaches proposed for GUI testing can be used. For example, event-based models such as ESGs are used for testing of web service compositions, and, also, there are formalisms which can be alternative/complementary to the finite-state-machine-based approaches in fault-based testing.

6.1.2 Contracts will Become More Precise and Look Different

Further test generation techniques, which make use of the intuitive approaches to reduce and/or simplify test sequences and their generation, are considerable for contract-based GUI testing. The more GUI contracts exist, the costlier is their evaluation. This is because adding just one single contract doubles (in the worst case) the number of combinations of contracts to be tested. Thus, further techniques to reduce the evaluation complexity of large sets of GUI contracts can be helpful, such as partitioning of contracts that can be achieved by a hierarchical set of DTs, if GUI contracts are represented with DTs.

Another path for future work is to introduce formal semantics for GUI contracts, which may include refinement and inclusion operations on contracts. These operations aim to provide distinct means to express complex GUI behavior in terms of simpler behavior. Furthermore, a refinement enables specialization of contractual obligations and invariants of other contracts, whereas inclusion allows contracts to be composed from simpler contracts. With these contract operations, an approach for combining GUI components and testing them as a single unit can be defined and implemented. More formalism for GUI contracts is necessary to systematically discuss and justify the differences between GUI contracts and classical Meyer contracts with respect to preconditions, post-conditions, invariants, and inheritance mechanism.

Such formalism on GUI contracts may end up with a formal contract language for specification of GUIs with composition and inheritance mechanisms. Contracts written in this language can be converted to test cases and test oracles can be built automatically.

6.1.3 What About Automation of GUI Testing?

Model-based testing has been proven to bring advantage in terms of test effectiveness and test efficiency when testing graphical user interfaces. However graphical user interfaces have been evolving over time from systems where it was easily possible to test merely every interface in the past to more complicated systems. Today's challenge is to know what to test for in graphical user interfaces rather than how to test as not everything is testable anymore. Future graphical user interfaces are heading towards more intelligent systems with changing feedback behavior over time not only adapting to the user behavior but even predicting the user's behavior. Furthermore, graphical user interfaces will integrate voice interfaces utilizing speech processing (such as apple's Siri). The results of these more intelligent user interfaces will become unpredictable and bring the challenge how to test them if we don't know what to test for.

Model-based testing tends to generate a large amount of test cases which will become impractical even if executed automatically for future graphical user interfaces. On the contrary side testing needs to withstand the coverage of the exponential growth of graphical user interfaces comprising more and more features. Thus, testing graphical user interfaces will require even more emphasis to optimize and minimize test suites in the future. The goal is to find coverage of graphical user interface interactions and events by test instances of test inputs and test sequences with high defect detection likelihood. This would also mean that traditional test coverage criteria are probably not adequate any longer and have to be replaced by a more property or situation based test input generation. In addition, the automation of the test oracle would need a heuristic approach when testing complex graphical user interfaces to overcome their characteristics of unpredictable test outcomes. This would tend towards an oracle that would find suspicious deviations and would need to define some new qualitative measurements.

6.2 Summary

The present book chapter reviewed existing work on model-based GUI testing. Both the desirable and undesirable features of the system to be developed have been unified, leading to a holistic approach to testing. Notions of mutation analysis and testing have been used to refine this holistic view.

Event-based modeling has been favored as it is broadly accepted. Due to the use of models that can be represented as graphs, application of sound mathematical methods is enabled. Results and algorithms have been borrowed from graph theory, automata theory, and formal languages to construct test cases, optimize test suites, etc. Modeling with event sequence graphs has been exemplarily used without any loss of generality, because also other graph-based models can be adapted for enabling the application of such formal methods.

The focus was on modeling and test case construction that covered also optimization. An important issue of testing in the practice is the quantification of the test cases, that is, assigning real and symbolic values to them. This aspect has been studied involving contract-based testing, which entails augmenting the graphs by decision tables.

Because of the limitation of the expected size of a chapter not all aspects of GUI testing could be discussed, for example, test prioritization, or considering other semantic features than "follows" relation in ESG modeling, for example concurrency or causality. Follow-on limitations can be seen in the definition and usage of fault and coverage semantics. Consequently, there are severe theoretical barriers, necessitating further research to extend and generalize the introduced ideas and techniques, mostly caused by the explosion of states when taking, for example, concurrency into account [22].

Nevertheless, the authors hope that further research will enable more and comprehensive adoption of the approaches introduced in the practice.

References – Section 1

[1] J. A. Whittaker, Software's invisible users, Software, IEEE, 18(3), (2001) 84–88.
[2] L. White and H. Almezen, Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences, in Proc. Int. Symposium on Softw. Reliability Engineering ISSRE 2000, IEEE Comp. Press, (2000) 110-119.
[3] B. Korel, Automated Test Data Generation for Programs with Procedures, Proc. ISSTA '96, (1996) 209-215.
[4] A. M. Memon, M. E. Pollack and M. L. Soffa, Hierarchical GUI Test Case Generation Using Automated Planning, IEEE Trans. Softw. Eng. 27/2, (2001) 144-155.
[5] D. Hamlet, Foundation of Software Testing: Dependability Theory, Proc. Of ISSTA '96, (1994) 84-91.
[6] M.A. Friedman, J. Voas, Software Assessment, John Wiley & Sons, New York, 1995.

References – Section 2

[7] 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, (2001) 34-43.
[8] A.M. Memon, M.L. Soffa, M.E. Pollack, Coverage Criteria for GUI Testing, Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9), ACM, (2001) 256-267.
[9] 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, (2013) 48-59.
[10] F. Belli, M. Beyazıt, Exploiting Model Morphology for Event-Based Testing, IEEE Transactions on Software Engineering, vol. 41, no. 2, (2015) 113-134.
[11] L. White, H. Almezen, User-Based Testing of GUI Sequences and Their Interactions, Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001), IEEE Computer Society, (2001) 54-63.
[12] F. Belli, M. Beyazıt, A.T. Endo, A. Mathur, A. Simao, Fault Domain-Based Testing in Imperfect Situations - A Heuristic Approach and Case Studies, Software Quality Journal, vol. 23, no. 3, (2015) 423-452.
[13] R.K. Shehady and D.P. Siewiorek, A Method to Automate User Interface Testing Using Variable Finite State Machines, Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS 1997), IEEE Computer Society, (1997) 80-88.
[14] A.C.R. Paiva, N. Tillmann, J.C.P. Faria, R.F.A.M. Vidal, Modeling and Testing Hierarchical GUIs, the 12th International Workshop on Abstract State Machines (ASM 2005), (2005) 8-11.
[15] F. Belli, C.J. Budnik, A. Hollmann, Holistic Testing of Interactive Systems Using Statecharts, Journal of Mathematics, Computing & Teleinformatics (AMCT), vol. 1, no. 3, (2005) 54-64.
[16] F. Belli, A. Hollmann, Test Generation and Minimization with 'Basic' Statecharts, Proceedings of the 23rd ACM Symposium on Applied Computing (SAC 2008), ACM, (2008) 718-723.
[17] 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, (2011) 249-258.
[18] 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, (2012) 2211-2218.
[19] 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, (2008) 1-35.
[20] 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) 277-326.
[21] Private Communication: The idea of ESG extension, mainly reflecting Professor Ina Schieferdecker's idea, bases on early, unpublished discussions between her and Fevzi Belli, 2006.
[22] 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, (2012) 198-206.

[23] A.M. Memon, I. Banerjee, A. Nagarajan, GUI Ripping: reverse engineering of graphical user interfaces for testing, Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003), IEEE Computer Society, (2003) 260-269.

[24] 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, (2005) 884-896.

[25] 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, (2007) 396-405.

[26] 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) 333-342.

[27] 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, (2010) 121-130.

[28] E.F. Moore, Gedanken Experiments on Sequential Machines, Automata Studies, Annals of Mathematical Studies, vol. 34, Princeton University Press, (1956) 129-153.

[29] G.H. Mealy, A method for synthesizing sequential circuits, Bell Systems Technical Journal, vol. 34, Sep. (1955) 1045-1079.

[30] R.G. Hamlet, Testing Programs with the Aid of a Compiler, IEEE Transactions on Software Engineering, vol. SE-3, no. (1977) 279-290.

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

[32] T.A. Budd, A.S. Gopal, Program Testing by Specification Mutation, Computer Languages, vol. 10, no. 1, (1985) 63-73.

[33] M. Beyazıt, Exploiting Model Morphology for Event-Based Testing, PhD Thesis, University of Paderborn, 2014.

[34] 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, (2006) 9-18.

[35] M.K. Kwan, Graphic Programming Using Odd or Even Points, Chinese Math., vol. 1, no. 3, 1962, 273-277.

[36] J. Edmonds, E.L. Johnson, Matching, Euler Tours and the Chinese Postman, Mathematical Programming, vol. 5, no. 1, (1973) 88-124.

[37] 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. (1991) 1604-1615.

[38] 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, (2011) 212-221.

[39] P. Purdom, A Sentence Generator for Testing Parsers, BIT Numerical Mathematics, vol. 12, no. 3, (1972) 366-375.

[40] 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.

[41] 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, (2009) 130-135.

References – Section 3

[42] A. M. Memon, M. E. Pollack and M. L. Soffa, Automated Test Oracles for GUIs, SIGSOFT 2000, (2000) 30-39.

[43] M. Marré, A. Bertolino, Using Spanning Sets for Coverage Testing, IEEE Trans. on Softw. Eng. 29/11, (2003) 974-984.

[44] A. Gargantini and C. Heitmeyer, Using Model Checking to Generate Tests from Requirements Specification, *Proc. ESEC/FSE, 1999*. ACM SIGSOFT, (1999) 146-162.

[45] D.B. West, Introduction to Graph Theory, Prentice Hall, 1996.

[46] H. Thimbleby, The Directed Chinese Postman Problem, School of Computing Science, Middlesex University, London, 2003.

[47] Edger. W. Dijkstra, A note on two problems in connexion with graphs, Journal of Numerische Mathematik, Vol. 1, (1959) 269-271.

[48] R. K. Shehady and D. P. Siewiorek, A Method to Automate User Interface Testing Using Finite State Machines, in Proc. Int. Symp. Fault-Tolerant Comp. FTCS-27, (1997) 80-88.

[49] C. Wohlin, P. Runeson. *Experimentation in Software Engineering – An Introduction.* Kluwer Academic Publishers: 2000.

[50] F. Belli, N. Nissanke, Ch. J. Budnik, A Holistic, Event-Based Approach to Modeling, Analysis and Testing of System Vulnerabilities, Technical Report TR 2004/7, Univ. Paderborn (2004).

[51] Fevzi Belli, Christof J. Budnik, Test minimization for human-computer interaction, Appl. Intell. 26(2), (2007) 161-174.

[52] Christof J. Budnik, Fevzi Belli, Axel Hollmann, Structural Feature Extraction for GUI Test Enhancement, ICST Workshops, (2009) 255-262.

[53] Fevzi Belli, Christof J. Budnik, Lee White, Event-based modelling, analysis and testing of user interactions: approach and case study. Softw. Test., Verif. Reliab. 16(1), (2006) 3-32.

References – Sub-Chapter 4

[54] B. Meyer, Applying design by contract, Computer (Long. Beach. Calif)., vol. 25, no. 10, (1992) 40–51.

[55] J. M. Jazequel, B. Meyer, Design by contract: the lessons of Ariane, Computer, vol. 30, no. 1., (1997) 129–130.

[56] T. Tuglular, C. A. Muftuoglu, F. Belli, M. Linschulte, Model-Based Contract Testing of Graphical User Interfaces, IEICE Transactions on Information and Systems E98.D(7), (2015) 1297-1305.

[57] C. D. T. Cicalese, S. Rotenstreich, Behavioral specification of distributed software component interfaces, Computer (Long. Beach. Calif)., vol. 32, no. 7, (1999) 46–53.

[58] W. Zheng, G. Bundell, Test by contract for uml-based software component testing, in Proceedings of IEEE International Symposium on Computer Science and its Applications, Washington, DC, USA, (2008) 377–382.

[59] I. Ciupa, A. Leitner, Automatic testing based on design by contract, in Proceedings of Net. ObjectDays (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World), (2005) 545–557.

[60] L.C. Briand, Y. Labiche, H. Sun, Investigating the use of analysis contracts to improve the testability of object-oriented code, Software Pract Exper; 33(7), (2003) 637–672.

[61] B. K. Aichernig, Contract-based mutation testing in the refinement calculus, Electr. Notes Theor. Comput. Sci., vol. 70, no. 3, (2002) 281.

[62] P. Madsen, Testing By Contract—Combining Unit Testing and Design by Contract, in The Tenth Nordic Workshop on Programming and Software Development Tools and Techniques, (2002), see also: http://www.itu.dk/people/kasper/NWPER2002/papers/madsen.pdf

[63] P. Guerreiro, Simple Support for Design by Contract in C++, In Proc. of the 39th Int. Conf. and Exhibition on Technology of Object-Oriented Languages and Systems, July 29 - August 03 2001, IEEE, Washington, DC.

[64] R. Plösch, Design by Contract for Python, IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference (APSEC97/ICSC97), HongKong, December 2-5, 1997.

[65] R. Heckel, M. Lohmann, Towards contract-based testing of web services, Electron. Notes Theor. Comput. Sci., vol. 116, (2005) 145–156.

[66] G. Dai, X. Bai, Y. Wang, F. Dai, Contract-Based Testing for Web Services, Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International, vol. 1., (2007) 517–526.

[67] R. Heckel, M. Lohmann, Towards contract-based testing of web services, Electr. Notes Theor. Comput. Sci., (2005) 145-156.

[68] E. Valentini, G. Fliess, E. Haselwanter, A framework for efficient contract-based testing of software components, 29th Annual International Computer Software and Applications Conference (COMPSAC'05), vol. 1, (2005) 219–222.

[69] D. Xu, W. Xu, M. Tu, Automated Generation of Integration Test Sequences from Logical Contracts, Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International. (2014) 632–637.

[70] IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, 1990.

[71] B. Baudry, Y. Le Traon, and J. Jezequel, Robustness and diagnosability of OO systems designed by contracts, Seventh International Software Metrics Symposium, METRICS 2001, Proceedings, (2001) 272–284.

[72] T. Tuglular, C. A. Muftuoglu, F. Belli, M. Linschulte, Event-Based Input Validation Using Design-by-Contract Patterns, ISSRE 2009, 20th International Symposium on Software Reliability Engineering, Mysuru, Karnataka, India, (2009) 195-204.

[73] T. Tuglular, F. Belli, M. Linschulte, Input Contract Testing of Graphical User Interfaces, Int. J. Softw. Eng. Knowl. Eng., vol. 26, no. 02, (2016) 183–215.

[74] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, D. D. Edwards, Artificial intelligence: a modern approach, vol.2, Prentice Hall, Englewood Cliffs, USA, 1995.

[75] R. Helm, I. M. Holland, D. Gangopadhyay, Contracts: Specifying behavioral compositions in object-oriented systems, vol. 25. no. 10. ACM, (1990) 169-180.

References – Section 5

[76] Hartmann, A., AGEDIS Model based Test Generation Tools, AGEDIS Consortium, 2002.

[77] Wiki Comparison of GUI testing tools, https://en.wikipedia.org/wiki/Comparison_of_GUI_testing_tools, July 2014

[78] Gartner Report on Software Test Automation Tools, 2016

[79] D. Kelly, Software test automation and the product life cycle, MacTech, 13, 1999.

[80] B. Marick, When should a test be automated, (1998): http://www.exampler.com/testing-com/writings/automate.pdf

[81] E. Hendrickson, Making the right choice: The features you need in a gui test automation tool, STQE Magazine, (2003) 20–25.

[82] B. Pettichord, Success with test automation, Quality Week, (2001): https://www.prismnet.com/~wazmo/succpap.htm

[83] A. Pretschner, J. Philipps, 10 Methodological Issues in Model-Based Testing, Model-Based Testing of Reactive Systems, Lecture Notes in Computer Science, vol. 3472, (2005) 281-291.

[84] qftestJUI homepage. At URL: http://www.qfs.de

[85] jfcUnit homepage. At URL: http://jfcunit.sourceforge.net

[86] ISO/IEC/IEEE 29119 Testing Standard: http://www.softwaretestingstandard.org/index.php

[87] Ranorex Test Automation homepage. At URL: http://www.ranorex.com/

[88] HP Unified Functional Testing (UFT) homepage. At URL: http://www8.hp.com/de/de/software-solutions/unified-functional-automated-testing/

[89] Froglogic Squish homepage. At URL:: https://www.froglogic.com/squish/

[90] Sekuli Script homepage. At URL: http://www.sikuli.org/

[91] Conformiq Software: Conformiq Test Generator homepage. At URL: http://www.conformiq.com

[92] Smarttesting CertifyIt homepage. At URL: http://www.smartesting.com/en/certifyit/

[93] ESG Test Suite Designer (ESG-TSD) homepage. At URL: http://download.ivknet.de/

[94] M. Hübner, I. Philippow, and M. Riebisch, Statistical usage testing based on UML, Proceedings of the 7th World Multiconferences on Systemics, Cybernetics and Informatics, Orlando, FL, USA, Jul 27, 2003.

[95] R. K. Swain, P. K. Behera, and D. P. Mohapatra, Minimal TestCase Generation for Object-Oriented Software with State Charts, arXiv preprint, arXiv:1208.2265, 2012.

[96] F. Belli, M. Linschulte, T. Tuglular, Karar Tablosu Destekli Olay Sıra Çizgeleri Temelli Sınama Durum Üretim Aracı, Proceedings of the 10th Turkish National Software Engineering Symposium, Canakkale, Turkey, (2016) 408-413.