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: Belli, F., Budnik, Ch. J., White, L., Event-based Modeling, Analysis and Testing of User Interactions: Approach and Case Study, Software Testing, Verification and Reliability, Vol. 16, No. 1, pp. 3-32, John Wiley & Sons, Ltd,

Event-based modelling, analysis and testing of user interactions: approach and case study

Fevzi Belli[1,*,†], Christof J. Budnik[1] and Lee White[2]

[1]*Department of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Warburger Strasse 100, D-33098 Paderborn, Germany*
[2]*Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH 44106-7071, U.S.A.*

SUMMARY

With the growing complexity of computer-based systems, their graphical user interfaces have also become more complex. Accordingly, the test and analysis process becomes more tedious and costly. This paper introduces a holistic view of fault modelling that is carried out as a complementary step to system modelling, enabling a scalability of the test process, and providing considerable potential for automation. Event-based notions and tools are used to generate and select test cases systematically. The elements of the approach are illustrated and validated by a case study. This paper does not claim to introduce a novel theoretic approach; rather, it makes use of graph-theoretic results for a practical and simple, but nevertheless powerful, view of modelling, analysis and testing of graphical user interfaces.

KEY WORDS: modelling/analysis/testing of graphical user interfaces; event sequence graphs

1. INTRODUCTION

In terms of behavioural patterns, the relationship between a system and its environment, e.g. the user, can be described as *proactive*, *reactive* or *interactive*. In the proactive case, the system generates the stimuli which evoke the activity of its environment. In the reactive case, the system behaviour evolves through its responses to stimuli generated by its environment. Most human–machine interfaces are nowadays interactive in the sense that the user and the system can be both proactive and reactive.

When developing interactive systems, construction of the user interactions deserves special care, and should be handled separately because it requires different skills, and maybe different techniques than construction of common software. This fact has been recognized very early, defining a

*Correspondence to: Fevzi Belli, Department of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Warburger Strasse 100, D-33098 Paderborn, Germany.
†E-mail: belli@upb.de

User Interaction Management System that can be independent of the application, graphics package, etc. [1]. The present paper will focus on the *user interface* (UI), especially on specification and validation aspects of the system behaviour and its collaboration with the user. Today UIs are usually implemented graphically; this paper will concentrate on *graphical user interfaces* (GUIs); UI and GUI will be used interchangeably.

The design part of the UI development needs a good understanding of the user and his/her needs, while the implementation part requires familiarity with the technical equipment, e.g. programming platform, language, etc. [2,3]. Testing requires both a good understanding of user requirements, and familiarity with the technical equipment. This paper is about UI testing; more precisely, testing of the software that implements the UI. To some extent, analysis aspects will also be covered as testing and analysis usually belong together. Apart from GUI testing, the approach can be deployed for user-centred testing and analysis of any human–computer system including reactive systems, e.g. Internet applications, vending machines, CD players, etc. [4].

When observing an interactive human–machine system, depending on the expectations of the user concerning the system behaviour, a distinction is to be made between *desirable* and *undesirable* situations, or *events*. The set of desirable events defines the system properties. Any deviation from the expected behaviour amounts to an undesirable situation. Alongside the desirable system behaviour, for a *holistic* approach, it is necessary to also consider the undesirable behaviour of the system as a complementary view.

Based on previous work by Belli [4] and by White *et al*. [5], this paper introduces a holistic view of fault modelling that requires an additional, complementary step to system modelling. Thus, both the desirable and undesirable behaviour of the system is specified at the same level of design granularity. In other words, intended and unintended usage scenarios will be generated to check both that system behaviour is in compliance with the user's expectations and that faults are handled properly. This is one of the important contributions of the approach presented in this paper. For modelling, the approach merges the notions of *state*, *input*, and *output* to events. This event-based view is more convenient for a tester because he or she is not primarily interested in internal states of the system under test (SUT), but rather in events that can externally be observed, perceived and evaluated. This view has several advantages which are explained in the next section.

Testing is usually carried out by applying test cases to the SUT. While constructing *test cases*, one generally has to produce meaningful test *inputs* and then to determine the expected system *outputs* for these inputs. Accordingly, to generate test cases for a UI, one first 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 behaviour as an output by means of well-defined test input. In a broader sense, the test object is the SUT; the objective of the test is to gain confidence in the SUT.

Test inputs of the GUI usually represent sequences of GUI object activities and/or selections that will operate interactively with the objects (*interaction sequences* (ISs) [6]; see also [7], *event sequences* (ESs)). Such an interactive sequence is *complete* (CIS, see [6]), if and only if it eventually produces the desirable system response. A major problem is the unique distinction between correct and faulty events (the *oracle problem*). The approach represented here exploits the concept of CIS to handle the oracle problem in an effective manner. This is another contribution of the paper.

During the testing of a SUT by a set of test cases, the test results can be satisfactory; yet this success is limited to the exercised test cases. Therefore, for the quality judgement of the SUT one needs

further quantitative arguments for the adequacy of the test cases and test process, usually achieved by well-defined coverage criteria [8]. The most well known coverage criteria are based either on structural issues of the SUT (*implementation-oriented/white-box testing*), or its behavioural, functional description (*specification-oriented/black-box testing*), or both, if both implementation and specification are available (*hybrid/grey-box testing*). Further, additional criteria are needed to control the test process and judge the effectiveness of the test cases, and to determine the point in time at which to stop testing (the *test termination problem*).

The approach introduced in this paper is specification oriented. For a systematic, scalable [9] generation and selection of test sequences, and accordingly, for test termination, the notion of *coverage of event sequences of a given length* of the graphical representation of the SUT is introduced. This graphical representation is called an *event sequence graph* (ESG) and has several advantages. First, it enables an incremental refinement (see also [10,11]) of the specification which may be rough and rudimentary at the beginning, or even non-existent. Second, the approach can also be deployed in implementation-oriented testing, e.g. using the implementation (source code) as a concise description of the SUT as the ultimate specification, and its control flow diagram (see also [12]).

The merits of the approach are summarized as follows.

- It is holistic in the sense that both intended (i.e. correct) and unintended (i.e. exceptional) usage scenarios are generated. This enables checking both that system behaviour is in compliance with the user's expectations and that faults are handled properly.
- The notion of complete interaction sequences is systematically explored to handle the oracle problem in an effective and scalable manner.
- Especially in cases where no system specification and design specification are available for test generation, the approach suggests event sequence graphs which can be simply and incrementally created from user manuals or system descriptions. This is very convenient especially for testing off-the-shelf commercial systems.
- The approach can also be deployed both in specification-oriented testing and implementation-oriented testing.
- It is not limited to GUI testing and can be deployed for any user-centred testing.

These are aspects that have not often been uniformly handled by a single, integrating approach as achieved by this paper.

Section 2 summarizes related work. Theoretical background, fault modelling and test concepts are introduced in Section 3. The test platform and SUT of the case study are described in Section 4, the results of which are discussed in Section 5. Section 6 concludes the paper, summarizing the results and planned future work.

The results of the work presented here are novel, including the validation and empirical analysis, and lead to an integrated, holistic methodology to generate and select test cases in the sense of a meaningful criterion.

2. RELATED WORK

Methods based on finite-state automata (FSA) have been used for almost four decades for modelling and validation of complex systems, e.g. for conformance testing [13–15], as well as for specification

and testing of system behaviour [14,16,17]. Shehady and Siewiorek [18] and White *et al.* [5] both introduced an FSA-based method for GUI testing, and the latter work included a convincing empirical study to validate the approach. This model has been extended to consider not only the desirable situations, but also the undesirable ones [4]. This strategy is quite different from the combinatorial ones, e.g. pairwise testing, which requires that for each pair of input parameters of a system, every combination of these parameters' valid values must be covered by at least one test case. This is, in most practical cases, not feasible [19,20].

A similar fault model as given by Belli [4] is used in the mutation analysis and testing approach which systematically and stepwise modifies the SUT using *mutation operations* [21]. This approach has been well understood, is widely used and, thus, has become quite popular. Although originally applied to implementation-oriented unit testing, mutation operations have also been extended to be deployed at more abstract, higher levels, e.g. integration testing, state-based testing, etc. [22]. Once applied to the graphical representation of a SUT, mutation operations can be viewed as elements of the complementing steps of the event sequence graphs, as introduced in Section 3. Such operations have also been independently proposed by other authors, e.g. 'state control faults' for fault modelling by Bochmann and Petrenko [13], or for 'transition-pair coverage criterion' and 'complete sequence criterion' by Delamaro *et al.* [22]. However, the latter two notions have been precisely introduced by Belli [4] and by White and Almezen [6], respectively, prior to the work of Offutt *et al.* [17], where they also appeared.

Another state-oriented approach, based on the traditional SCR (Software Cost Reduction) method, has been described by Gargantini and Heitmeyer [23]. This approach uses model checking [24] (which can identify negative and positive scenarios) to generate test cases automatically from formal requirements specifications, using well known coverage metrics for test case selection. A different approach for GUI testing has been recently published by Memon *et al.* [25,26], which deploys methods of knowledge engineering to generate test cases, test oracles, etc., and to handle the test termination problem: from a knowledge engineering point of view, the testing of a GUI system represents a typical *planning* problem that can be solved using a goal-driven strategy [26]. Given a set of operators, an initial state and a goal state, the planner is expected to produce a sequence of operators that will change the initial state to the goal state. For the GUI testing problem described above, this means that the test sequences have to be constructed dependent upon both the desirable, correct events and the undesirable, faulty events.

There have also been other GUI testing papers from this research group. Memon *et al.* [27] have also written a more general paper on GUI testing that emphasizes coverage criteria involving events and event sequences. They decompose the GUI system into components and emphasize a hierarchy of interacting components in their generated tests. Thus, the tests can be viewed as a hierarchy as well. They have provided a case study showing the correlation between this coverage criterion and that of the underlying code coverage. A more recent paper [28] has shown how a more streamlined GUI testing method can be used for regression testing. Both approaches, i.e. those of Memon *et al.* and of Gargantini and Heitmeyer, use some heuristic methods to cope with the state explosion problem.

The approach presented in this paper is event based and has certain affinities with the finite-state-based approaches. First, finite, sequential processes are modelled. Second, directed graphs, i.e. ESGs, are used for modelling; these are related to state-transition diagrams of finite-state automata. The advantage of the event-based approach introduced in this paper stems from its simplicity of modelling for users, especially those from industry, perhaps with little experience of automata theory.

This fact enables broad acceptance of the approach as has been experienced with partners in practice [4]. Apart from the simplification of the modelling process, there are also advantages from the theoretical side as the following examples might demonstrate.

To construct test sets for the coverage of event sequences, well known algorithms for solving the transition tour problem of conformance testing, e.g. the approach of Aho *et al.* to handle the Chinese Postman Problem [29–31], have been utilized. However, these algorithms in their original version might not be appropriate for handling GUI testing problems, because the complexity of the optimization problem for GUI testing is considerably lower, as summarized in Section 3, based on the work of Belli [4].

Furthermore, for testing GUIs, existing methods of conformance testing, e.g. the W method of Chow [14], could be considered. There are, however, significant differences between the goals and assumptions of the W method and the approach presented in this paper. First, conformance testing generates tests to detect faults clearly defined by some hypotheses, e.g. transition errors, missing, or extra states. The approach in this paper generates test cases to *cover* the ESG. Second, the W method assumes two models are available, i.e. the system model and the design model. These two models are then compared. The first model is required to be correct, the second one is to be checked against the correct model. The approach in this paper supposes the existence of only one model that describes the correct behaviour of the system. This model is exploited to generate tests to check the real, implemented SUT, i.e. the SUT is tested against its specification. Third, conformance testing assumes the system under consideration behaves deterministically and is completely specified by a finite-state automaton. These assumptions are not necessarily fulfilled by GUIs; therefore, they are not made by the approach presented in this paper.

3. THEORETICAL BACKGROUND

As already mentioned, this work uses event sequence graphs (ESGs) to represent the system behaviour and, moreover, the facilities from the user's point of view while interacting with the system. Basically, an *event* is an externally observable phenomenon, such as an environmental or a user stimulus, or a system response, punctuating different stages of system activity. It is clear that such a representation disregards the detailed internal behaviour of the system, which is given by means of its different states and, hence, an ESG is a more abstract representation compared with, for example, a state transition diagram (STD) or finite-state automaton (FSA) [32,33]. In the following subsections, the notions used in the approach are formally introduced.

3.1. Preliminaries

Definition 1. An *event sequence graph* $ESG = (V, E)$ is a directed graph where $V \neq \emptyset$ is a finite set of *vertices* (*nodes*), $E \subseteq V \times V$ is a finite set of *arcs* (*edges*), and $\Xi, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in \Xi$, and $\gamma \in \Gamma$, called *entry* nodes and *exit* nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $\langle \xi, v_0, \ldots, v_k \rangle$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $v \neq \xi, \gamma$.

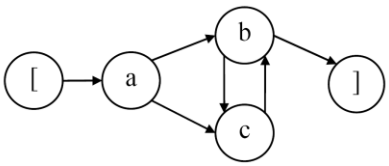


Figure 1. An ESG with a as entry and b as exit and pseudo vertices $[\ ,\ ]$.

$\Xi(ESG)$, $\Gamma(ESG)$ represent the entry nodes and exit nodes of a given ESG, respectively. To mark the entry and exit of an ESG, all $\xi \in \Xi$ are preceded by a pseudo vertex $[\notin V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex $] \notin V$.

The semantics of an ESG is as follows. Any $v \in V$ represents an event. For two events $v, v' \in V$, the event v' must be enabled after the execution of v if and only if $(v, v') \in E$.

The operations on identifiable components of the UI are controlled and/or perceived by input/output devices, i.e. elements of windows, buttons, lists, checkboxes, etc. Thus, an event can be a user input or a system response; both of them are elements of V and lead interactively to a succession of user inputs and expected desirable system outputs.

Definition 2. Let V, E be defined as in Definition 1. Then any sequence of vertices $\langle v_0, \ldots, v_k \rangle$ is called an *event sequence* (ES) if $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$.

Note that the pseudo vertices $[\ ,\ ]$ are not included in the ESs. An $ES = \langle v_i, v_k \rangle$ of length 2 is called an *event pair* (EP). Accordingly an *event triple* (ET), *event quadruple* (EQ), etc. can be defined.

Example 1. For the ESG given in Figure 1, $V = \{a, b, c\}$, $\Xi = \{a\}$, $\Gamma = \{b\}$, and $E = \{(a, c), (a, b), (b, c), (c, b)\}$. Note that arcs from pseudo vertex $[$, and to pseudo vertex $]$, are not included in E.

Furthermore, α (*initial*) and ω (*end*) are functions to determine the initial vertex and end vertex of an ES, e.g. for ES $= \langle v_0, \ldots, v_k \rangle$, the initial vertex and the end vertex are $\alpha(ES) = v_0$, $\omega(ES) = v_k$, respectively. For a vertex $v \in V$, $N^+(v)$ denotes the set of all *successors* of v, and $N^-(v)$ denotes the set of all *predecessors* of v. Note that $N^-(v)$ is empty for an entry $\xi \in \Xi$, and $N^+(v)$ is empty for an exit $\gamma \in \Gamma$.

Finally, the function l (*length*) of an ES determines the number of its vertices. In particular, if $l(ES) = 1$ then $ES = \langle v_i \rangle$ is an ES of length 1.

Note that the pseudo vertices $[$ and $]$ are not considered in generating any ESs. Neither are they considered in determining the initial vertex, end vertex, and length of ESs.

Example 2. For the ESG given in Figure 1, *bcbc* is an ES of length 4 with the initial vertex b and end vertex c.

Definition 3. An *ES* is a complete *ES* (or, it is called a *complete event sequence*, CES), if $\alpha(ES) = \xi \in \Xi$ is an entry and $\omega(ES) = \gamma \in \Gamma$ is an exit.

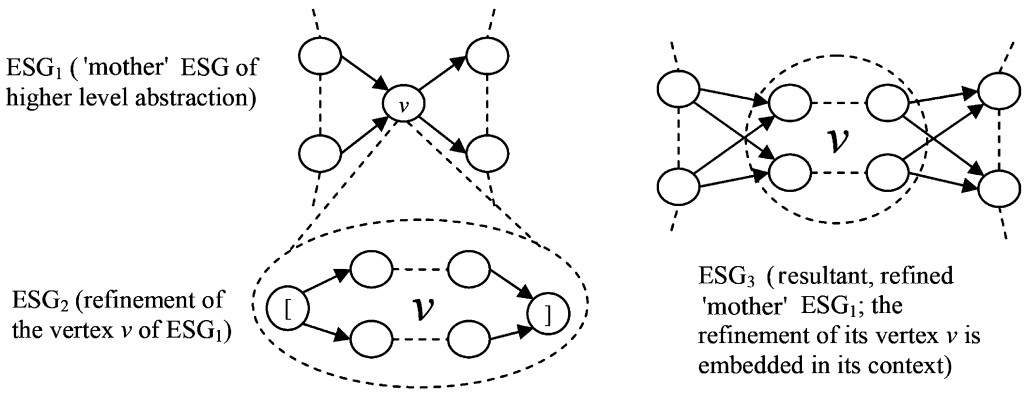


Figure 2. Refinement of a vertex v and its embedding in the refined ESG.

Example 3. *acb* is a CES of the ESG given in Figure 1. CESs represent *walks* from the entry of the ESG to its exit realized by the form

(*initial*) *user inputs* $\rightarrow$ (*interim*) *system responses* $\rightarrow \cdots \rightarrow$ (*final*) *system response*

Note that a CES may invoke no interim system responses during user–system interaction, i.e. it may consist of consecutive user inputs and a final system response.

Definition 4. Given an ESG, say $ESG_1 = (V_1, E_1)$, a vertex $v \in V_1$, and an ESG, say $ESG_2 = (V_2, E_2)$, then replacing v by ESG_2 produces a *refinement* of ESG_1, say $ESG_3 = (V_3, E_3)$ with $V_3 = V_1 \cup V_2 \backslash \{v\}$, and $E_3 = E_1 \cup E_2 \cup E_{\mathrm{pre}} \cup E_{\mathrm{post}} \backslash E_{1\mathrm{replaced}}$ ($\backslash$ is the set difference operation), wherein $E_{\mathrm{pre}} = N^-(v) \times \Xi(ESG_2)$ (connections of the predecessors of v with the entry nodes of ESG_2), $E_{\mathrm{post}} = \Gamma(ESG_2) \times N^+(v)$ (connections of exit nodes of ESG_2 with the successors of v), and $E_{1\mathrm{replaced}} = \{(v_i, v), (v, v_k)\}$ with $v_i \in N^-(v)$ and $v_k \in N^+(v)$ (replaced arcs of ESG_1).

As Figure 2 illustrates, *every* predecessor of vertex v of the ESG of higher level abstraction points to the entries of the refined ESG. By analogy, *every* exit of the refined ESG points to the successors of v. The refinement of v in its context within the original ESG of higher level abstraction contains no pseudo vertices *[* and *]* because they are only needed for the identification of entries and exits of the ESG of a refined vertex.

Example 4. In Figure 3 the refinement of the vertex a of ESG_1 is given as ESG_2. ESG_3 is the resulting refinement of ESG_1. Note that the pseudo vertices *[* and *]* of ESG_2 are not included in ESG_3.

More precisely, ESG_1 is given as $V_1 = \{a, b, c\}$, $E_1 = \{(a, b), (a, c), (b, c), (c, b)\}$. In the refinement, i.e. ESG_2 of a, the predecessors and successors are $N^-(v) = \{\,\}$, $N^+(v) = \{b, c\}$ and the refinement of ESG_2 is given by $V_2 = \{a_1, a_2, a_3\}$, $E_2 = \{(a_1, a_2), (a_1, a_3), (a_2, a_3), (a_3, a_2)\}$, $\Xi(ESG_2) = \{a_1\}$ and $\Gamma(ESG_2) = \{a_2, a_3\}$. The resultant ESG_3 is represented by

$$V_3 = V_1 \cup V_2 \backslash \{v\} = \{a, b, c\} \cup \{a_1, a_2, a_3\} \backslash \{a\} = \{b, c, a_1, a_2, a_3\}$$

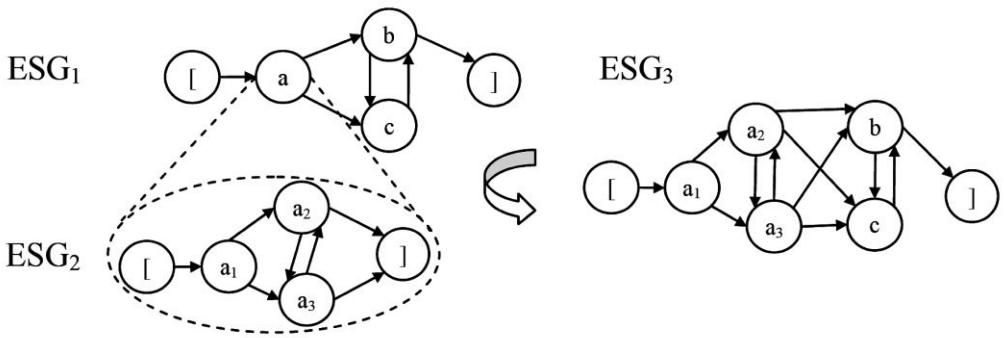


Figure 3. A refinement of the vertex a of the ESG given in Figure 1.

and

$$
\begin{aligned}
E_3 &= E_1 \cup E_2 \cup E_{\text{pre}} \cup E_{\text{post}} \backslash E_{1\text{replaced}} \\
&= \{(a,b),(a,c),(b,c),(c,b)\} \cup \{(a_1,a_2),(a_1,a_3),(a_2,a_3),(a_3,a_2)\} \cup \{\,\} \cup \{(a_2,b),(a_2,c), \\
&\quad (a_3,b),(a_3,c)\} \backslash \{(a,b),(a,c)\} \\
&= \{(b,c),(c,b),(a_1,a_2),(a_1,a_3),(a_2,a_3),(a_3,a_2),(a_2,b),(a_2,c),(a_3,b),(a_3,c)\}
\end{aligned}
$$

3.2. Fault model and test terminology

System malfunctions are manifest in the form of *failures*, thus affecting the ability of the system to perform its functions. A failure can be realized, or triggered, by a *fault* as an incorrect, thus an undesirable, event during the execution of the system. A fault can eventually be traced back to a human action, or inaction, leading to an incorrect and, thus, an undesirable result. Though there is a cause–effect relationship [34] between them, the terms 'error' and 'fault' are often used synonymously as the cause of a 'failure'. The purpose of any test effort is to force an error/fault (lying at the root of the above chain) to show up as a failure, i.e. as a situation that cannot be hidden from the environment, especially from the user of the system.

Faults are differentiated in this paper by two categories, as also described by White and Almezen [6]: *defects* are serious departures from specified behaviour; *surprises* are user-recognized departures from expected behaviour. A surprise behaviour is not explicitly indicated in the specification of the UI; it should, however, be perceived by some users as a disturbing or disappointing behaviour of the system. However, it cannot be emphasized strongly enough that for UI systems, because of sensitive usability issues, many surprises might well be considered more serious than some defects.

The approach introduced in this paper assumes that there is no user error, i.e. upon a faulty user input the system has to inform the user, and, wherever possible, point him or her properly in the right direction in order to reach the anticipated desirable situation. Due to this requirement, a complementary

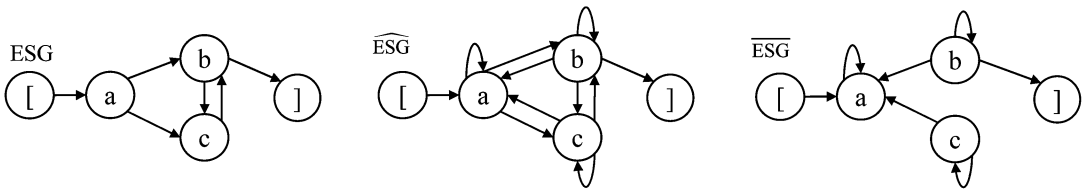


Figure 4. ESG of Figure 1, its completion $\widehat{ESG}$ and inversion $\overline{ESG}$, with $\overline{ESG} = \widehat{ESG} \backslash ESG$.

view is necessary to consider potential user errors in the modelling of the system (see also the works of Gaudel [35] and Koufareva *et al.* [36]).

Definition 5. For an $ESG = (V, E)$, its *completion* is defined as $\widehat{ESG} = (V, \hat{E})$ with $\hat{E} = V \times V$.

Definition 6. The *inverse* (or *complementary*) ESG is then defined as $\overline{ESG} = (V, \bar{E})$ with $\bar{E} = \hat{E} \backslash E$.

Figure 4 illustrates the definition of $\widehat{ESG}$, which can be systematically constructed in three steps.

- Add arcs in the opposite direction wherever only one-way arcs exist.
- Add self-loops to vertices wherever none exist.
- Add two-way arcs between vertices wherever no arcs connect them. Note that they are drawn bi-directional.

$\overline{ESG}$ (the inversion of the ESG) consists of arcs that will be added to the ESG to construct the $\widehat{ESG}$ (completion of the ESG).

Definition 7. Any EP of the $\overline{ESG}$ is a *faulty event pair* (*FEP*) for the ESG.

Example 5. *ca* of the given $\overline{ESG}$ in Figure 4 is a FEP.

Definition 8. Let $ES = \langle v_0, \ldots, v_k \rangle$ be an event sequence of length $k + 1$ of an ESG and $FEP = \langle v_k, v_m \rangle$ a faulty event pair of the corresponding $\overline{ESG}$. The concatenation of the ES and FEP then forms a *faulty event sequence* $FES = \langle v_0, \ldots, v_k, v_m \rangle$.

Example 6. For the ESG given in Figure 4, *bca* is an FES of length 3.

Definition 9. An FES is *complete* (or, it is called a *faulty complete event sequence*, *FCES*) if $\alpha(FES) = \xi \in \Xi$ is an entry. The ES as part of a FCES is called a *starter*.

Note that Definition 9 explicitly points out that a FCES does not finish at an exit, unlike a CES that must finish at an exit.

Example 7. For the ESG given in Figure 4, the FEP *ca* of the $\overline{ESG}$ can be completed to form the FCES *acbca* by using the ES *acbc* as a starter. Note that the *⌈* is not included in the FCES as it is a pseudo vertex.

n := number of the functional units (modules) of the system that fulfill a well-defined task

`length` := required length of the test sequences

`FOR` function `1 TO n DO`
`BEGIN`
    Generate the corresponding ESG and $\overline{ESG}$
    `FOR k:=2 TO length DO`
    `BEGIN`
        Cover all `ES`s of length `k` by means of `CES`s
    `END`
    Cover all `FEP`s by means of `FCES`s
`END`
Apply the test set to the SUT
Observe the system output to determine whether a correct system response or a faulty event occurs

Algorithm 1. Test process.

The starter *acbc* in Example 7 is arbitrarily chosen, and hence the variation in length of an FCES is always attributable to starters prior to this special FEP under consideration. The result is then FCESs of various lengths. Thus, the 'length' in the test process primarily relates to the CESs.

3.3. Test process

As already mentioned in Section 1, a major problem in testing is the determination of correct and faulty situations upon inputs (the oracle problem [25,37]). The approach introduced in this paper uses event sequences, more precisely CESs and FCESs, as test inputs. If the input is a CES, the SUT is supposed to accept this input and produce a desirable behaviour that is well defined prior to testing; thus the test succeeds. Accordingly, if a FCES is used as a test input, the SUT is supposed to refuse this input and produce a warning. The latter case represents an exception that must be properly handled by the system in order to avoid undesirable failures.

Definition 10. A *test case* is an ordered pair of an input and an expected output of the SUT. Any number of test cases can be extended to a *test set*.

The test process is summarized by the given Algorithm 1. The approach assumes that the system is modularly structured, e.g. having n units. Each of these units fulfills a task that can be modelled by an ESG.

The coverage-oriented test process of the approach leads to test cases which exercise the specified functions of the implemented SUT with the goal of covering these functions. This coverage must be, of course, economical in terms of the number of test cases. Therefore, a *stopping rule* for test case generation is needed. This is given in Algorithm 1 by the coverage of the event sequences of increasing length, which is determined by analysing the model.

3.4. Test configuration and test cost

The number and length of the event sequences are the primary factors that influence the cost of the test process. In order to compare the costs of different test configurations by a case study (see the next section), a precise definition of these notions is necessary.

Definition 11. A *test configuration* is defined as a quintuple $(l_{\max}, \#ES_i, l_{\text{cov}}, \#CES, \beta)$ with:

- $l_{\max}$ as the given maximum length of the ESs to be covered;
- $\#ES_i$ as the number of ESs of length i, with $i = 2, 3, \ldots, l_{\max}$;
- l_{cov} as the sum of lengths of CESs to cover all ESs of up to a given maximum length $l_{\max}$;
- $\#CES$ as the number of the CESs; and
- $\beta \in \mathbb{R}$ as the weight factor for conducting multiple tests.

These determine the *test costs for ESs* as

$$\Psi_{ES} := \beta \cdot \sum_{i=2}^{l_{\max}} \#ES_i + \sum_{i=2}^{l_{\max}} \#ES_i \cdot i$$

and *test costs for CESs* as

$$\Psi_{CES} := \beta \cdot \#CES + l_{\text{cov}}$$

One could consider the test costs of a test configuration to depend only on the number of tests; however, this would neglect the length of the tests and the costs of restart/undo before conducting a new test. The formulae Ψ_{ES} and Ψ_{CES} have been introduced to avoid this oversimplification.

The cost formulae Ψ_{ES} and Ψ_{CES} in Definition 11 have two terms. The first term determines the maximum number of ESs (in Ψ_{ES}) and the maximum number of CESs (in Ψ_{CES}); the latter is the maximum number of tests to be run. Thus, this first term reflects the test costs caused by restarting the system before initiating another test (*test multiplicity*). However, a single test can cover several ESs, even of different length. To adjust to a specific situation, the tester can vary the weight factor β. Typically, β has the value 1 if the tester has no hint about the multiplicity of a planned series of tests. The weight factor can be greater than 1 in order to reflect a situation with disproportionate costs for restarting the system before a new test. The value of β can also be zero if the costs of the restart process are negligible.

The second term in the cost formulae for Ψ_{ES} and Ψ_{CES} determines the total length of the test sequences (ESs and CESs) that must be run which contribute the other part of the costs.

For the deployment of these and Ψ_{CES} in the case study (Section 4.1), the assumption is made that each event and every restart have the same test costs, i.e. $\beta = 1$.

4. CASE STUDY

In this case study different test strategies are generated by varying the length and number of the event sequences to be covered, and the type of the complete event sequences to cover them, i.e. CES or FCES. Combinations of different variations lead to different test configurations and test costs. The benefit of a test is measured by the number and severity of faults that have been revealed at increasing cost.

Based on the notions of test configuration and test costs in Definition 11, a rudimentary cost–benefit analysis enables the tester to justify the efficiency of the approach, i.e. to answer the following questions.

- Is there a strong correlation between the length/number of the test cases and the number of faults revealed?
- Is there a strong correlation between the kind of tests (based on CES versus FCES) and the number of faults revealed?
- Is there a strong correlation between the length/number of test cases and the severity of the faults revealed?
- Is there a strong correlation between the kind of tests (based on CES versus FCES) and the severity of the faults revealed?

To sum up, the objective of the case study 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 will be collected and evaluated by means of experiments carried out in accordance with the principles of software experimentation [38,39].

4.1. Software under test

For the case study, *RealJukebox* (*RJB*) has been selected, more precisely the basic, English version of RJB 2 (Build: 1.0.2.340) of RealNetworks. There are several reasons why RJB has been selected to be the SUT. First, RJB is a commercial, popular application that is widely well known and accepted by a great variety of users. Second, the selected SUT has been used over many years in different languages and 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 the SUT avoided studying an 'alpha' version of a no-name product for the case study with the present approach.

Figure 5 represents the main menu of the RJB of RealNetworks that is a personal music management system. The user can build, manage, and play his or her individual digital music library on a personal computer. At the top level, the GUI has a pull-down menu with the options *File*, *Edit*, etc. that invoke other components. These sub-options have further sub-options, etc. There are still more window components which can be used to traverse through the entries of the menu and sub-menus, creating many combinations and accordingly, many applications.

In the absence of a manufacturer's system specification, namely, a functional description of the RJB, the help facilities and the handbook of the RJB are used to produce the references for construction of the test cases and test scripts, based on CESs as desirable events. Those functions describe the steps as to how to reach situations the user wants, i.e. desirable events in terms of *system functions* (responsibilities). For this case study, 12 different functions of the SUT (Table I) were identified.

4.2. Test platform and test configuration

RJB was tested on two different PCs with different processors in order to detect and filter likely permanent or transient errors within the hardware and/or system software in any one of

Figure 5. GUI of RJB.

Table I. System functions as responsibilities of the system to interact with the user.

1. Play and record a CD or track	7. Visualization
2. Create and play a playlist	8. Skins
3. Edit playlists and/or autoplaylists	9. Screen sizes
4. View lists and/or tracks	10. Different views of windows
5. Edit a track	11. Find music
6. Visit the sites	12. Configure RJB

these test environments. An assumption was made that it is very unlikely for the same system error to occur in both PCs.

By means of running the tests on two different platforms, random errors were excluded that cannot be reproduced. Thus, the reported errors are permanent ones and have been detected on both platforms (and not on only one stochastically). Furthermore, they are reproducible in an effective manner, provided that the system requirements specified in Table II are fulfilled.

The studied platforms are described in Table II. Both these platforms fulfill the system requirements of the manufacturer; thus, compatibility and conformance problems can be excluded. Significant characteristics were taken into account by considering the different options of run mode that are listed in the system description. The options differ from each other in the RJB settings:

Table II. Different test systems and their equipment.

	System requirements of the manufacturer	System 1 (test platform 1)	System 2 (test platform 2)
Operating system	Windows NT with Service Pack 4	Windows NT with Service Pack 4	Windows NT with Service Pack 4
Computer processing speed	Intel Pentium® 200 MHz	Intel Pentium® 266 MHz	Intel Pentium® 233 MHz
RAM (available)	32 MB	64 MB (9 MB)	64 MB (16 MB)
Graphics	16 bit colour video card (800 × 600 resolution)	32 bit colour video card (1024 × 768 resolution)	32 bit colour video card (1024 × 768 resolution)

`AutoPlay` activated/deactivated, `AutoRecord` activated/deactivated. Any inconsistencies that might occur during the testing have been carefully analysed, taking adjusted settings of the RJB into account.

4.3. Modelling the UI of RJB

As a SUT, RJB was used intensively and different options were investigated in order to improve understanding of the SUT in a stepwise fashion. At the same time, ESGs were produced and incrementally extended in terms which started at a very rudimentary level. Thus, the system was studied and the ESGs refined in accordance with Definition 4. Each of the further desirable events defines a system function that must be completed and/or refined and represented in an ESG refined accordingly. This modelling effort avoided a 'trial and error' way of random testing and enabled detection of many intricate faults as the results depict in Section 5. In the following, the modelling process is summarized for one of the system functions listed in Table I.

For representing user interactions, the vertices of the ESG are interpreted as the operations on identifiable objects that can be perceived and controlled by input/output devices. Thus, an event can be a user input as well as a system response that interacts with the next user input, triggering a new output. The set of symbols of an ESG, which label the vertices, will be interpreted here as an event set.

In accordance with Figure 5, the ESG in Figure 6 represents the top-level GUI to enable the interaction `Play and Record a CD or Track` via the main menu given in Figure 5. The user can load a CD, select a track and play it. He or she can then change the mode, replay the track, or remove the CD, load another one, etc. Figure 6 illustrates all sequences of user–system interactions to realize likely operations the user might launch when using the system.

The ESGs of Figure 7 refine the vertices of the graph depicted in Figure 6. The refinement `Select track` describes the alternative ways to select a track. `Play track` describes the variety of the functions similar to those encountered in an ordinary cassette player. The sub-graph `Mode` describes the different ways to control the operation, i.e. playing of the tracks. Any of the design levels can be used to generate CESs, and thus test cases. As refined levels include more information, more test cases can be generated by analysing those refined ESGs.

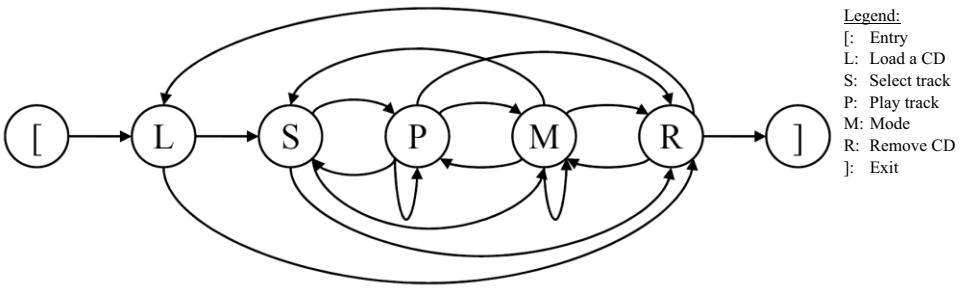


Figure 6. The system function `Play and Record a CD or Track` represented as an ESG.

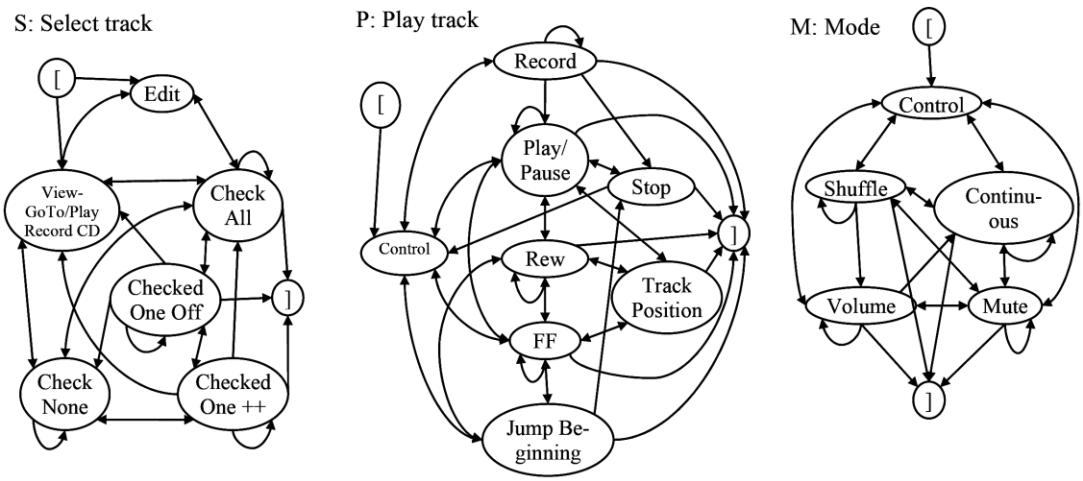


Figure 7. Refinement of the vertices *S*, *P* and *M* of the ESG in Figure 6.

Each of the vertices of an ESG in Figures 6 and 7 represents user inputs which interact with the system, leading eventually to events as system responses that are expected, i.e. correct situations. Thus, each edge of the ESG represents a pair of subsequent legal events which was defined as an event pair.

First, the ESG and its complement of the system function `Play and Record a CD or Track` (Figure 8) is analysed. Example 8 lists the corresponding EPs of Figure 8.

Example 8. *LS*, *LR*, *SP*, *SM*, *SR*, *PS*, *PP*, *PR*, *PM*, *MP*, *MS*, *MM*, *MR*, *RL*, *RM*. As a next step of the test case construction, CESs are generated as test inputs for the example sketched in Figure 8.

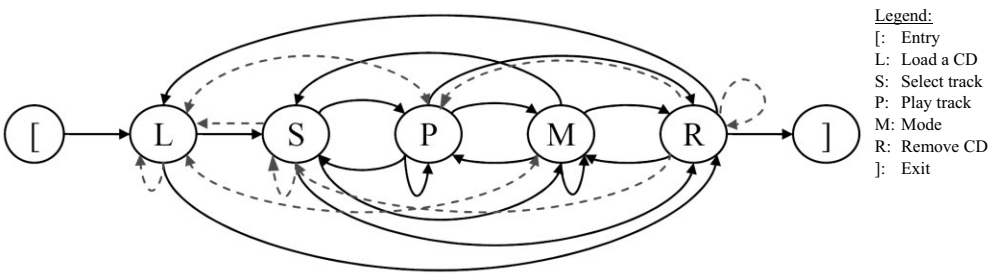


Figure 8. ESG and its complement $\overline{ESG}$ of the system function `Play and Record a CD or Track`.

Example 9. *LSR*, *LR*, *LSPR*, *LSMR*, *LSR*, *LSPSR*, *LSPPR*, *LSPR*, *LSPMR*, *LSMPR*, *LSMSR*, *LSMMR*, *LSMR*, *LRLR*, *LRMR*. It is evident that some CESs have been generated multiple times using different EPs, e.g. *LSR* by extending the *LS* to the right (by *R*) and *SR* to the left (by *L*). Elimination of this redundancy leads to Example 10.

Example 10. *LSR*, *LR*, *LSPR*, *LSMR*, *LSPSR*, *LSPPR*, *LSPMR*, *LSMPR*, *LSMSR*, *LSMMR*, *LRLR*, *LRMR*. The set of CESs given in Example 10 ensures that all EPs will be covered; it is, however, not optimized as a set of test inputs.

The dashed lines in Figure 8 represent the FEPs of the function `Play and Record a CD or Track` of the RJB; they will be listed in Example 11. Note here again some arcs are bi-directional, e.g. *LP* and *ML*.

Example 11. *LL*, *SL*, *LP*, *PL*, *LM*, *ML*, *SS*, *RP*, *RR*, *RS*. Furthermore, each FEP can be extended to a complete faulty event sequence FCES given as Example 12 by determining appropriate starters which start at the entry. The starters in Example 12 are of minimal length to save test costs.

Example 12. *LL*, *LP*, *LM*, *LSL*, *LSPL*, *LSPML*, *LSS*, *LSPMRP*, *LSPMRR*, *LSPMRS*.

4.4. Tool support

To avoid tedious and error-prone manual work, the tool 'GATE' (Generation and Analysis of Test Event sequences) was built. The tool accepts the adjacency matrix of the ESG and automatically generates test inputs of the required length for a given ESG, i.e. *ES*, *FES*, *CES*, and *FCES*. As a first step, all ESs of a given length are produced. The tool can also generate CESs of a given length which cover all ESs. Furthermore, GATE determines the effectiveness of a given test case in terms of covered ESs. If no length is explicitly given, the tool constructs a CES of minimal length that covers all ESs.

Figure 9 depicts the main frame window for test generation. For the example given in Figure 6, GATE generates CESs of up to length 8 and ESs of up to length 4. The tester requires in this example that loops be run twice. Furthermore, the weight factor β is set to 1.0 (see Definition 11). Figure 10 depicts the output of a test case set which is analysed in Figure 11.

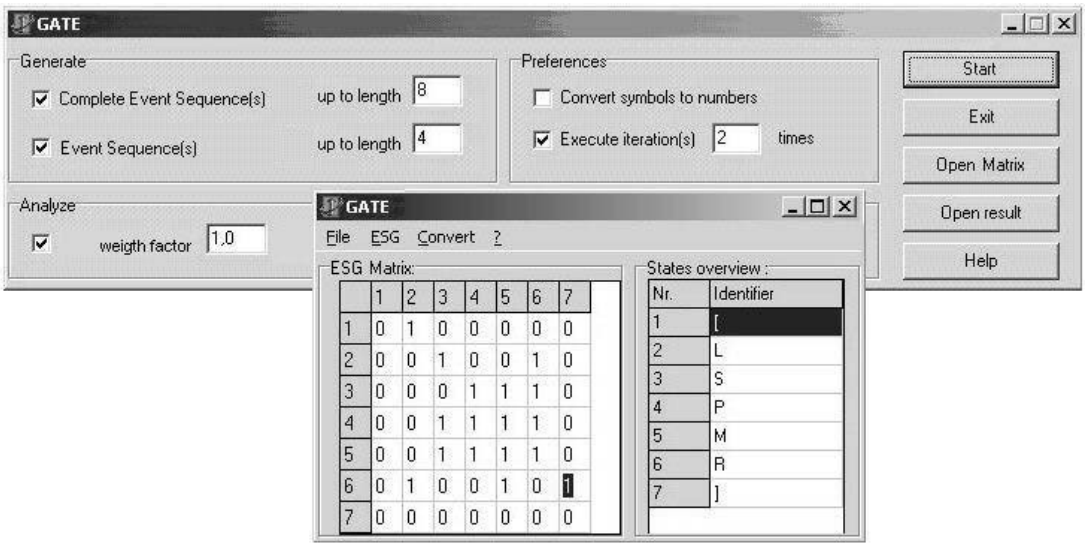


Figure 9. Test tool for test input generation for the ESG of Figure 6 (given by its adjacency matrix).

GATE generates ESs (or FESs) and CESs (or FCESs) of different length, depicted in the right and left halves of Figure 10, respectively.

The results of the analysis of the ESG given in Figure 6 are summarized in Figure 11. The goal of the analysis is the determination of the coverage ratio of ESs (column 2) by appropriate CESs (column 1) in increasing order of their length. As an example, the first line (middle part of Figure 11) indicates that 100% of EPs (as ESs of length 2) are covered by CESs of length 5. These CESs however, only cover 57% of the ETs (as ESs of length 3). CESs of length 7 cover 100% of ESs of length up to 4. The bottom part of Figure 11 calculates for ESs and CESs the number of test cases in accordance with Definition 11. These ESs and CESs have been previously determined by the tool GATE as demonstrated in Figure 10. Finally, the test costs are given in the bottom right column.

Another time-consuming and error-prone activity is given by tests that include long, routine processes (*test scripts*) that are to be manually carried out by the tester. To perform such processes automatically during the case study, the commercial tool WinRunner was deployed. This tool supports not only the execution of test scripts, but also their generation in capture/replay mode (see Figure 12).

To enable a unique identification and retrieval of the GUI objects, it is necessary to start with the same main frame window. All models that are produced in the case study exit in such main frame windows which, in turn, enable a consecutive execution of the test cases. Accordingly, the generated CESs realize meaningful applications and avoid the situations where several windows are simultaneously kept open. In the case when a FCES is executed, the error message of WinRunner is reported and the system is backtracked to the start state.

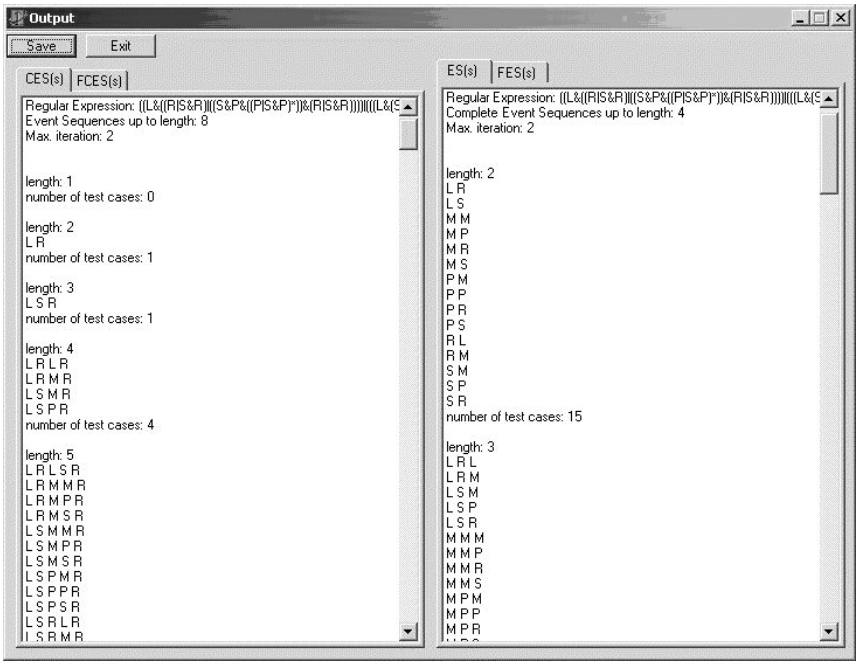


Figure 10. Test cases generated for ES/FES and CES/FCES.

5. RESULTS AND ANALYSIS OF THE CASE STUDY

This section summarizes the results of the case study, starting with a brief overview of the tests generated and executed, and faults detected, including some examples. Test characteristics are determined and dependencies between them are made transparent. Finally, the findings are analysed and discussed to come to a rationale for applying the approach in practice.

5.1. Overview

The number of CESs and FCESs depends on the extent of the connectivity of the ESG under consideration. In the extreme case, when there is a bi-directional edge between every pair of vertices and self loops at every vertex, only CESs can be generated, i.e. the set of FCESs is empty.

Table III depicts the generated number of test cases for each function from Table I sorted by length.

In Table IV different values are assigned to the weight factor β to take different levels of costs of resuming the test process after performing a test into account (see Section 3.4, Definition 11). It becomes apparent that the test cost reduction increases with:

- increasing length of covered ES; and
- increasing weight factor.

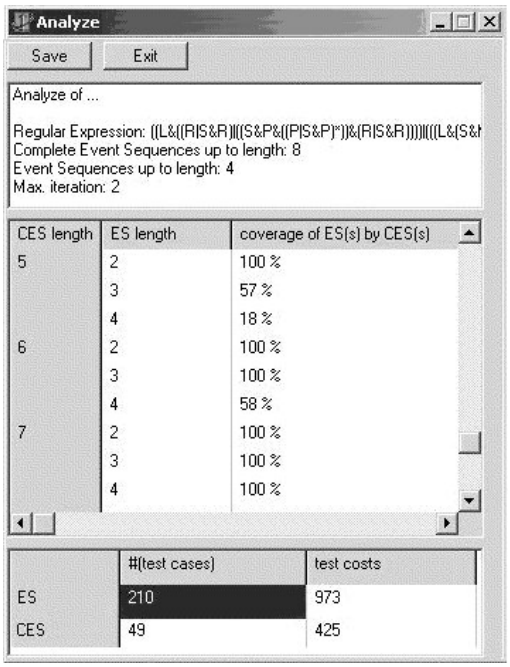


Figure 11. Analysis of the generated test cases.

Table III. Test cases by the functions of Table I.

	Function											
Length of covered ES	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
2	78	108	41	145	126	167	49	52	7	14	92	35
3	425	358	81	398	358	162	159	114	10	30	206	157
4	1918	1238	115	793	885	23	456	199	10	66	493	740

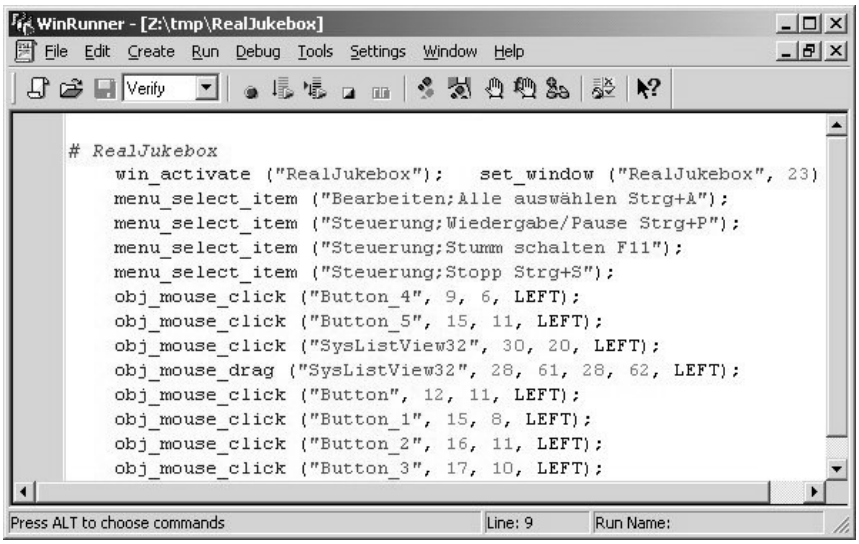


Figure 12. Generated test script of a CES for the example RealJukebox.

Table IV. Test costs of ES and CES and their percentage reduction.

Length of covered ES	Weight factor β	Test cost Ψ_{ES}	Test cost Ψ_{CES}	Reduction (%)
2	0	938	826	11.94
	1	1407	985	29.99
	2	1876	1144	39.02
3	0	4706	3384	28.09
	1	5962	3854	35.36
	2	7218	4324	40.09
4	0	16 414	10 865	36.00
	1	19 341	11 890	38.52
	2	22 268	12 915	42.00

This occurs if CESs are used as test inputs instead of ESs because CESs frequently cover several ESs of a given length by one single test (see the last column of Table IV).

An overview of faults found subject to their sequence length is illustrated in Table V, which summarizes the number of detected faults in the ESs to be covered by the total test cases from Table III. It clearly shows that the detected faults by covering ESs of length 2 is a subset of the detected faults by covering ESs of length 3, which is, in turn, a subset of the detected faults by covering ESs of length 4.

Table V. Overview of faults found, test cases and their distribution over sequence length covered.

Length of covered ES	Detected faults	Additional faults
2	44	+44
3	56	44 + 12
4	68	56 + 12

Table VI. Additional faults by function.

Length of covered ES	Function											
	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
2	28	0	5	2	1	3	3	0	0	0	2	0
3	3	0	0	2	1	0	3	2	0	0	1	0
4	1	0	3	2	0	0	1	3	0	0	1	1

Table VI shows the number of additional faults detected in different functions in terms of the length of ESs that are to be covered. In Table VI, it is interesting to note that many faults were detected in function 1, and yet not so many faults were detected in any of the other functions. Most notable were functions 2, 9 and 10, in which no faults were detected at all. First of all, function 1, `Play and Record a CD or Track`, corresponds to very high complexity, as can be seen from the tests required in Table III and also in Figures 6 and 7. Yet by far the most faults (28) were detected by just ESs of length 2; no other comparable number of faults is seen by any other function. Also many other functions required more tests of length 2, but did not discover nearly this many faults (only five faults for function 3 is the next largest number of faults for ESs of length 2). Function 2 has nearly as many tests as function 1; but since this is logic just to create a playlist, there are clearly not as many faults as in actually playing a CD or track. As for functions 3, 9 and 10, they are of much lower complexity, and require far fewer tests than function 1.

It is worth mentioning that an additional 12 transient faults have been found during the testing procedure. These faults have been detected randomly and on only one of the test systems; thus they could not be reproduced on both test platforms (see Table II). Therefore, they are not included in Tables V and VI, which contain only faults that can be reproduced anytime on either platform exercising the same test case.

5.2. Detected faults

Table VII arbitrarily extracts some of the detected faults. The fault reproduction process is very simple. As an example, in order to reproduce the fault number 1, one starts with the ‘Control’ option of the

Table VII. Excerpt from the list of faults revealed by testing the system function number 1 of Table I.

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 the Pause state and the 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 play-back although the selected tracks have not been completely played.	*Play/Pause FF FF FF*

main menu of the RJB (see Figure 13) and subsequentially pushes the Record button and then the FF button. In Figure 13, the dashed line with the label '1' uniquely identifies this sequence of actions. The other faults of Table VII can be reproduced in the same way.

Note that the faults numbered 2 and 5 are not included in Figure 13 as they are detected via the completed ESGs of Figures 7(*S*) and 7(*M*). Due to space constraints, these completed ESGs are not included in this paper.

5.3. Cost efficiency

Table VIII summarizes the number of test cases, their length and the corresponding faults, classified as surprises and defects (for definitions, see Section 3.2).

The number of defects detected by test cases of length 3 and 4 increases notably 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 14 shows the detected defects and surprises against test length. Generally there are more defects than surprises, specifically for test lengths of 2 and 3. Only at length 4 are there an equal number of defects and surprises (see also Table VIII).

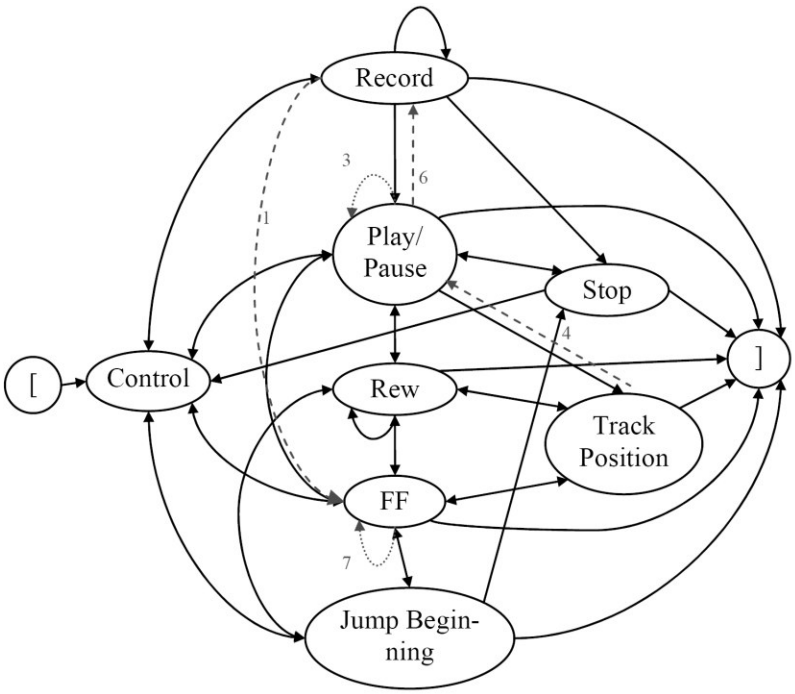


Figure 13. FEP revealed faults in the sub-graph 'Play track' of Figure 7.

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

Length of covered ES	Test cases	Detected faults by CES	Detected faults by FCES	Total number of detected faults	Surprises	Defects	Fault per test case (efficiency)
2	914	24	20	44	16	28	4.8×10^{-2}
3	2458	$24 + 7$	$20 + 5$	56	$16 + 8$	$28 + 4$	2.3×10^{-2}
4	6936	$31 + 4$	$25 + 8$	68	$24 + 10$	$32 + 2$	0.9×10^{-2}

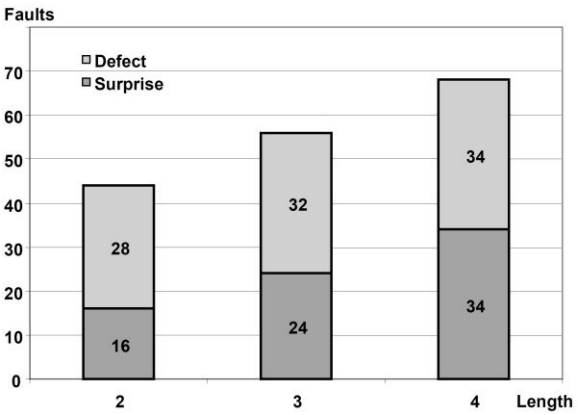


Figure 14. Detected defects and surprises depending on the event sequence length.

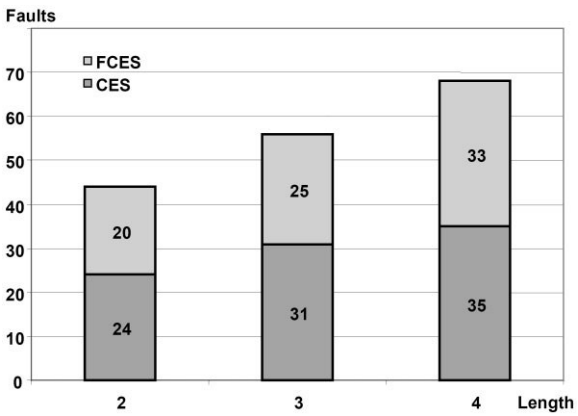


Figure 15. Detected faults by CES and FCES.

It is also interesting to note that FCES-based test cases really did deliver approximately the same number of faults as the CES-based test cases (Figure 15). Again, it is concluded that the reason is a poor design strategy for the GUI which concentrates on the realization of the desirable events and neglects the handling of the undesirable ones.

Figure 16 combines and refines the results found in Figures 14 and 15. 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 FCESs of length 4 in relation to only six based on FCESs of length 2! Thus, a clear tendency can be observed that an increasing number and length of

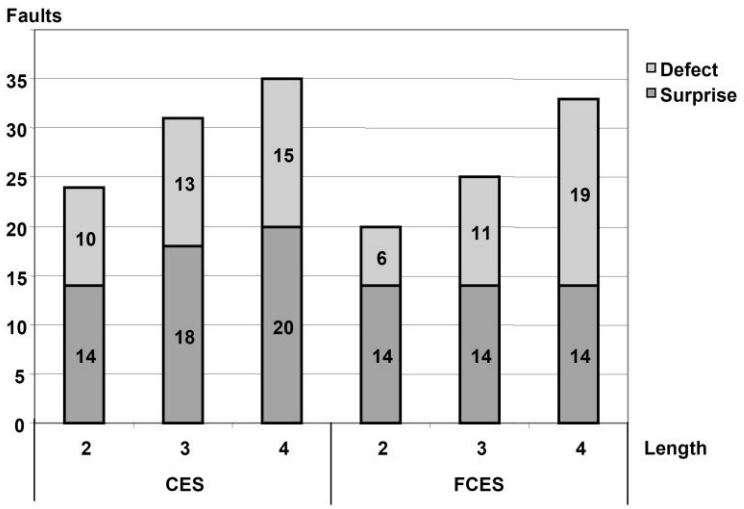


Figure 16. Defects and surprises based on CES/FCES, depending on the event sequence length.

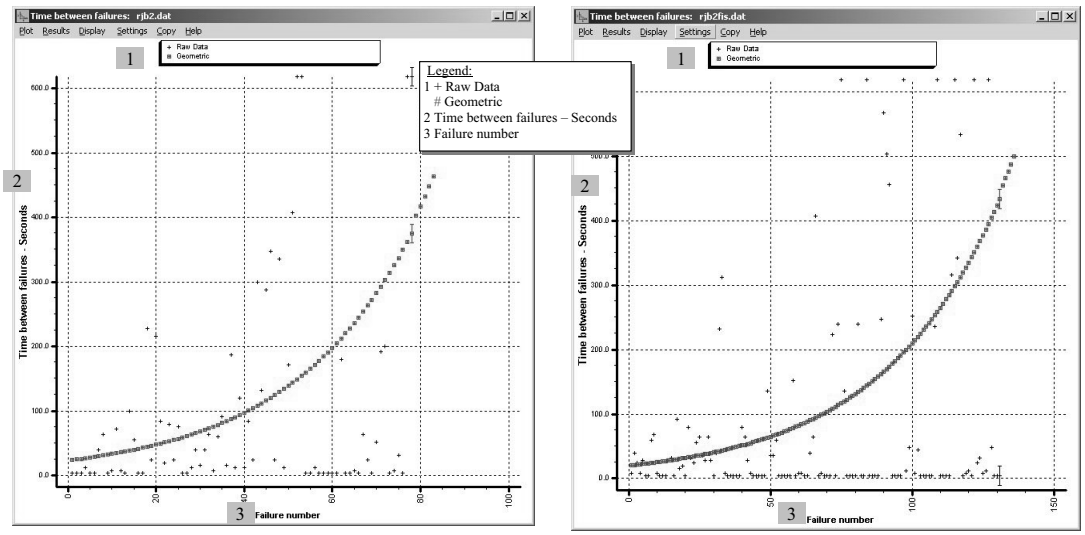
CES-based and FCES-based test cases lead to the detection of an increasing number of defects. Note, however, that Figure 16 does not consider the number of necessary tests, i.e. test costs.

5.4. Time between failures

Upon the detection of a fault, it was analysed and categorized (surprise/defect, ES-/FES-based, length). Then the system was restarted, i.e. recovered from the faulty state and the test process continued. The manual part of the case study was carried out by two testers (students); this took about 2 weeks, 6 h a day. Thus, about 432 000 s were spent in executing 78 611 test cases, resulting in 5.5 s per test. These figures, together with the approximate time points of the detected faults, have been used as input to the reliability tool CASRE [40], selecting several models. It was found that the reliability models Musa-Okumoto, Musa-Basic, Geometric, and Jelinski-Moranda correlated well with the data. The best 'goodness of fit' could be achieved by a Geometric model, the results of which are depicted in Figures 17(a) and 17(b) for the CESs and FCESs, respectively.

Note that the objective is not primarily to determine the reliability level of the SUT, but to produce approximating curves for cumulative numbers of faults over time. The number of faults in Figures 17(a) and 17(b) is greater than the real number because the repetitive occurrence of faults could not be eliminated. The SUT could also not be corrected before testing it further.

In spite of these imprecise data, a tendency can be clearly observed by all invoked models: the test cases based on CESs are considerably more cost-efficient than the ones based on FCESs, i.e. the test costs per detected fault based on CESs are lower, indicating the fact that the SUT has a rudimentary exception handling mechanism, even if not complete.



(a) (b)

Figure 17. (a) Time between failures based on the CESs of Figure 13. (b) Time between failures based on the FCESs of Figure 13.

5.5. Interpretation and discussion of the results—lessons learned

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

- The RJB that has been tested in this paper is a product that has matured over 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 VIII and Figures 16–18 clearly show the fact that CES-based tests of length 2 are the most cost-effective in this approach, i.e. they detect faults at the lowest cost 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 in the number of 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; see Table VIII). This finding can be explained by analysing 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 that are connected with each other, the larger is the number of tests to cover ESs that are longer than 2.

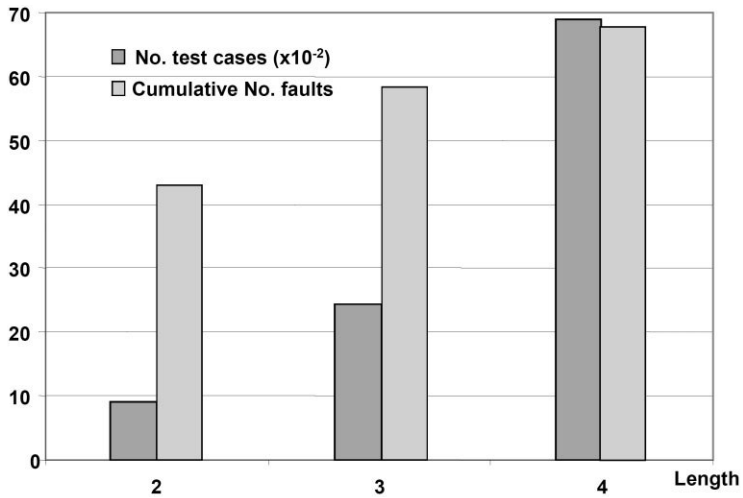


Figure 18. The cumulated number of detected faults in relation to the number of test cases.

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 those which are FCES-based.

Based on these observations, it is strongly recommended to start 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 test cases. 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, 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.

6. CONCLUSIONS AND FUTURE WORK

This work is intended to extend, refine and validate the approaches introduced previously [4,5,41,42] by taking into account not only the desirable behaviour, but also the undesirable behaviour of a GUI. This could be seen as the most important contribution of the present work, i.e. testing GUIs not only through exercising them by means of test cases which show that the GUI is working properly during routine operation, but also exercising potentially illegal events to verify that the GUI behaves satisfactorily in exceptional situations. Moreover, having an exact terminology and an appropriate

formal framework, one can now precisely scale the test process, justifying the cumulative costs that must be in compliance with the test budget.

The fault model in this paper has been intentionally kept simple and differs from that of Chow [14]: states, inputs and outputs of a FSA have been merged into the vertices that are interpreted as events. ESs represent user interactions; an ES is complete (CES) if it produces desirable, well-defined and safe system functionality.

The case study demonstrated the applicability of the approach, revealing the fact that CES-based tests of length 2 are very cost-effective, having a high capability of detecting errors. Thus, the coverage of the ESs and FESs of length 2 is a good starting point that can be followed by ESs of higher length. The case study also demonstrated that the approach could efficiently be deployed, supported by a chain of tools that are partly commercially available and partly self-developed.

The next step is to apply the approach to analyse and test *safety* features [43]; in this case the risks originate from within the system due to potential failure and its spillover effects causing potentially extensive damage to its environment. Another goal for future work is to design a defense action, which is an appropriately enforced sequence of events, to prevent faults that could potentially lead to such failures.

Problems of cost reduction are subjects of current research to extend the approach.

- Examine the value of the weight factor (β in Definition 11) where the reduction is minimum.
- Optimize the test sets (CESs and FCESs to cover ESs of a given length) that are to be constructed with the algorithm 'Test Process' in Section 3.3.
- Consider the structure of the SUT to find heuristics that further reduce the test sets.

The introduced holistic approach, unifying the modelling of both the desirable and undesirable features of the system to be developed, enables the adoption of the concept 'design for testability' in software construction; this concept was initially introduced in the 1970s [44] for hardware. It is hoped that further research will enable the adoption of this approach in more current modelling tools such as statecharts [3,30,45], UML [46,47], etc. There are, however, some severe theoretical barriers, necessitating further research to extend the algorithms developed in the ESG environment, mostly caused by the explosion of additional vertices and states while completing the ESG in order to take concurrency into account [48–50].

REFERENCES

1. Thomas JJ, Hamlin G (eds.). Graphical input interaction techniques: Workshop summary. *ACM Computer Graphics News* 1983; **17**(1):5–30.
2. Shneiderman B. *Designing the User Interface: Strategies for Effective Human–Computer Interaction* (3rd edn). Addison-Wesley: Reading, MA, 1998.
3. Horrocks I. *Constructing the User Interface with Statecharts*. Addison-Wesley: Reading, MA, 1999.
4. Belli F. Finite-state testing and analysis of graphical user interfaces. *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, Hong Kong, November 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 34–43.
5. White L, Almezen H, Alzeidi N. User-based testing of GUI sequences and their interactions. *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, Hong Kong, November 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 54–63.
6. White L, Almezen H. Generating test cases for GUI responsibilities using complete interaction sequences. *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE 2000)*, San Jose, CA, October 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000; 110–121.

7. Korel B. Automated test data generation for programs with procedures. *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA 1996)*, San Diego, CA, January 1996. ACM Press: New York, 1996; 209–215.
8. Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. *ACM Computing Surveys* 1997; **29**(4):366–427.
9. Dreyer J. *Program Segmentation for Controlling Software Testing and Analysis*. Shaker Verlag: Aachen, Germany, 1998.
10. Martin JC. *Introduction to Languages and the Theory of Computation* (2nd edn). McGraw-Hill: New York, 1997.
11. Holcombe M, Ipate F. *Correct Systems: Building a Business Process Solution*. Springer: Berlin, 1998.
12. Gossens S. Enhancing system validation with behavioural types. *Proceedings of the 7th International Symposium on High-Assurance Systems Engineering (HASE 2002)*, Tokyo, October 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 201–208.
13. Bochmann Gv, Petrenko A. Protocol testing: Review of methods and relevance for software testing. *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA 1994)*, Seattle, WA, August 1994. ACM Press: New York, 1994; 109–124.
14. Chow TS. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* 1978; **4**(3):178–187.
15. Sarikaya B. Conformance testing: Architectures and test sequences. *Computer Networks and ISDN Systems* 1989; **17**(2):111–126.
16. Parnas DL. On the use of transition diagrams in the design of a user interface for an interactive computer system. *Proceedings of the 24th ACM National Conference*, August 1969. ACM Press: New York, 1969; 379–385.
17. Offutt J, Liu S, Abdurazik A, Ammann P. Generating test data from state-based specifications. *Software Testing, Verification and Reliability* 2003; **13**(1):25–53.
18. Shehady RK, Siewiorek DP. A method to automate user interface testing using finite state machines. *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, Seattle, WA, June 1997. IEEE Computer Society Press: Los Alamitos, CA, 1997; 80–88.
19. Cohen DM, Dalal SR, Fredman ML, Patton GC. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 1997; **23**(7):437–444.
20. Tai K-C, Lei Y. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering* 2002; **28**(1):109–111.
21. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 1978; **11**(4):34–41.
22. Delamaro ME, Maldonado JC, Mathur AP. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering* 2001; **27**(3):228–247.
23. Gargantini A, Heitmeyer C. Using model checking to generate tests from requirements specifications. *Proceedings of the 7th European Software Engineering Conference (ESEC-7) and the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, Toulouse, France, September 1999 (*Lecture Notes in Computer Science*, vol. 1687). Springer: Berlin, 1999; 146–162.
24. Peled DA. *Software Reliability Methods*. Springer: New York, 2001.
25. Memon AM, Pollack ME, Soffa ML. Automated test oracles for GUIs. *Proceedings of the 8th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-8)*, San Diego, CA, November 2000. ACM Press: New York, 2000; 30–39.
26. Memon AM, Pollack ME, Soffa ML. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering* 2001; **27**(2):144–155.
27. Memon AM, Pollack ME, Soffa ML. Coverage criteria for GUI testing. *Proceedings of the 8th European Software Engineering Conference (ESEC-8) and the 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, Vienna, September 2001. ACM Press: New York, 2001; 256–267.
28. Memon AM, Banerjee I, Hashin N, Nagarajan A. DART: A framework for regression testing nightly/daily builds of GUI applications. *Proceedings of the 2003 International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, September 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 410–419.
29. Aho AV, Dahbura AT, Lee D, Uyar MÜ. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *IEEE Transactions on Communications* 1991; **39**(11):1604–1615.
30. Harel D, Naamad A. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(4):293–333.
31. Shen Y-N, Lombardi F, Dahbura AT. Protocol conformance testing using multiple UIO sequences. *IEEE Transactions on Communications* 1992; **40**(8):1282–1287.
32. Gill A. *Introduction to the Theory of Finite-State Machines*. McGraw-Hill: New York, 1962.
33. Gluschkow WM. *Theorie der Abstrakten Automaten*. VEB Verlag der Wissensch: Berlin, 1963.
34. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE STd. 610.12. 1990.
35. Gaudel M-C. Testing can be formal, too. *Proceedings of Theory and Practice of Software Development (TAPSOFT 1995)*, Aarhus, Denmark, May 1995 (*Lecture Notes in Computer Science*, vol. 915). Springer: Berlin, 1995; 82–96.

36. Koufareva I, Petrenko A, Yevtushenko N. Test generation driven by user-defined fault models. *Proceedings of the 12th IFIP International Workshop on Testing of Communicating Systems (IWTCS 1999)*, Budapest, Hungary, September 1999. Kluwer Academic: Norwell, MA, 1999; 215–236.
37. Hamlet D. Predicting dependability by testing. *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA 1996)*, San Diego, CA, January 1996. ACM Press: New York, 1996; 84–91.
38. Basili VR, Selby RW, Hutchens DH. Experimentation in software engineering. *IEEE Transactions on Software Engineering* 1986; **12**(7):733–743.
39. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic: Norwell, MA, 2000.
40. Lyu MR (ed.). *Handbook of Software Reliability Engineering*. McGraw-Hill: New York, 1996.
41. Eggers B, Belli F. A theory on analysis and construction of fault-tolerant systems (in German). *Proceedings, Informatik-Fachberichte*, Bonn, September 1984. Springer: Berlin, 1984; 139–149.
42. Belli F, Grosspietsch K-E. Specification of fault-tolerant system issues by predicate/transition nets and regular expressions: Approach and case study. *IEEE Transactions on Software Engineering* 1991; **17**(6):513–526.
43. Nissanke N, Dammag H. Design for safety in Safecharts with risk ordering of states. *Safety Science* 2002; **40**(9):753–763.
44. Williams TW, Parker KP. Design for testability: A survey. *IEEE Transactions on Computers* 1982; **31**(1):2–15.
45. Harel D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 1987; **8**(3):231–274.
46. Kim YG, Hong HS, Bae DH, Cha SD. Test cases generation from UML state diagrams. *IEE Proceedings—Software* 1999; **146**(4):187–192.
47. Briand L, Labiche Y. A UML-based approach to system testing. *Software and System Modeling* 2002; **1**(1):10–42.
48. Gutzeit Th. Testcase generation from statecharts to validate graphical user interfaces (in German). *Masters Thesis*, University of Paderborn, Angewandte Datentechnik, 2003.
49. Schneider FB. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 1990; **22**(4):299–319.
50. Raju SCV, Shaw AC. A prototyping environment for specifying, executing and checking communicating real-time state machines. *Software—Practice and Experience* 1994; **24**(2):175–195.