UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

FAKULTÄT FÜR
ELEKTROTECHNIK,
INFORMATIK UND
MATHEMATIK

Technical Reports and Working Papers

Angewandte Datentechnik (Software Engineering)
Prof. Dr.-Ing. Fevzi Belli

Test Generation Using Event Sequence Graphs

Fevzi Belli[1], Nimal Nissanke[2], Christof J. Budnik[1], Aditya Mathur[3]

[1]: Dept. of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Germany
[2]: Institute for Computing Research, BCIM, London South Bank University, London, UK
[3]: Department of Computer Science, Purdue University, West Lafayette, IN, USA

(Version 1.1, 05. September 2005)

Technical Report 2005/6 Version 1.1, 05. September 2005

D-33095 Paderborn, Pohlweg 47-49 Tel.:+49-5251/60-3447 Fax: /60-3246
http://adt.upb.de email: belli@adt.upb.de

Test Generation Using Event Sequence Graphs

Fevzi Belli[1], Nimal Nissanke[2], Christof J. Budnik[1], Aditya Mathur[3]

[1)]: Dept. of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Germany

[2)]: Institute for Computing Research, BCIM, London South Bank University, London, UK

[3)]: Department of Computer Science, Purdue University, West Lafayette, IN, USA

Abstract

An Event Sequence Graph (ESG) is a simple albeit powerful formalism for capturing the behavior of a variety of interactive systems that include real-time, embedded systems, and graphical user interfaces. A collection of ESGs is proposed as a model of an interactive system. This collection is used for the generation of tests to check for the correctness of system behavior in the presence of expected and unexpected input event sequences. The proposed test generation algorithm is customizable in the sense that it allows a tester to generate test sequences based on an evaluation of their cost of execution and the benefit derived. Two case studies, an empirical and an analytical, are reported. The empirical study is an assessment of the fault detection effectiveness of the approach. The analytical study is to demonstrate the power of ESGs in modeling and risk analysis.

Contents

1. Introduction

This paper presents a novel approach to modeling, analysis and testing of system behaviors, with respect to both correct and faulty behaviors, based on Event Sequence Graphs (ESGs). The approach focuses on (a) testing a system for correctness with respect to its behavioral requirements and for its robustness against incorrect inputs, and (b) analyzing the consequences under possible malfunction. It is based on two simple ideas. First, the desirable behavior of the user and the system under test (SUT) is modeled using a finite set of ESGs. Second, each ESG in this set is inverted algorithmically to obtain a formal representation of the undesirable user behavior and to work out the corresponding desired response by the SUT. The set of ESGs and their inversions are then used for test generation and for system malfunction analyses.

Complexity in behavioral patterns is an important source of faults, and hence failures, in computer-based systems. Such patterns express the relationship between the system and its environment, possibly including the user and other collaborating systems, and fall into three different categories: proactive, reactive, and interactive. ESGs apply to all three, though this work places some emphasis on interactive systems. ESGs focus on their externally observable behavior through discrete event-based models and provide a unified view of the interactions between the user stimuli and environmental actions, and system responses, though they are separated out later during the testing phase. ESGs differ from the finite-state based approaches in that they are based on a finite sequence of events, rather than states.

Our objective is to systematize the test generation process with a twin-track strategy. The first is to confine the scope of tests by targeting them at a given stage at a chosen system attribute such as user-friendliness or safety. This is achieved by an ordering of system states according to the risks posed to that attribute and to selecting tests that address specific threats. The second is to devise test plans where the tests are naturally ordered according to diminishing returns in terms of their cost-effectiveness. Technically, this is based on test length. This is possible because the tests are formulated in terms of sequences of non-faulty event pairs in the ESG model when testing the system for correctness of desirable (functional) features, and sequences of non-faulty event pairs in system ESG model followed a faulty event pair when testing for any undesirable outcome. Test space is explored using established algorithms to Chinese Postman Problem but modified with the aim of achieving greater efficiency. The algorithm establishes complete and minimal set of test cases that are necessary to exercise all event sequences of increasing lengths. Sequence length being a natural measure of the test costs, tests are conducted beginning from relatively low cost tests based on shorter sequences and proceeding onto more costly ones with longer sequences. Monitoring of the returns as the testing progresses thus provides a judicious basis for deciding when to terminate tests. The paper illustrates the approach using two case studies and demonstrates the effectiveness of the approach based on empirical results on actual tests.

In brief, the main contributions of this work are: (a) an ESG based formalism for the modeling and analysis of the behavior of discrete event, and sequential systems; (b) a test generation algorithm that takes an ESG as input and generates tests for testing the behavior of a SUT under expected and unexpected conditions, and (c) an evaluation of the feasibility and effectiveness of the proposed modeling and test generation schemes using two case studies. Compared to (Belli, 2001), where it and was first used for the study of user interactions, this work extends and refines it in following respects: (a) The present paper introduces a solid formal background for handling ESGs and modeling aspects. (b) Based on this background, algorithms are introduced and analyzed for test generation and optimization. (c) The fault model is extended in order to handle a broad variety of malfunctions and risks, qualitatively and quantitatively. (d) The scope of modeling is generalized so that now not only user interfaces can be modeled, but any kind of reactive or interactive systems. (d) Two case studies, one empirical and another one analytical, validate the approach and study its deployment in different environments, identifying and varying characteristic factors. (e) Tools that are necessary for an effective deployment of the approach are introduced and discussed. (e) Lessons learned from intensive and extensive application of the approach to industrial projects are summarized.

The remainder of this paper is organized as follows. Section 2 provides a summary of the related literature and establishes the relationship between finite-state automata (FSA) based models and ESG based models of system behavior. Section 3 is a rigorous introduction to ESGs. Section 4 sketches the test generation algorithm. Section 5 reports an empirical study that investigates the fault detection effectiveness and the cost of test generation; leaving the details of the study to Appendices 2 and 3. A study in Section 6 demonstrates the analytical power of ESG based modeling, while Section 7 provides an overview of a toolkit developed to support ESG based modeling. A discussion related to the current work and directions for further research appear in Sections 8 and 9, respectively. The terminology used in this paper is based on IEEE 610.12, ISO/IEC 9126 and IEC 60300-3-1.

2. Related Work

There is a vast body of work that addresses the problem of test generation for software systems. Here we review literature related to the modeling and generation of tests.

ESGs, with variations in terminology, have been used for modeling finite-state behavior since the work of Kleene (Kleene, 1958). Chhikara et al. use event sequence diagrams to study dynamic probabilistic risk analysis (Chhikara, 2000). Memon et al. use event flow graphs to model GUI event sequences as test cases (Memon, 2001). Event graphs and timed event graphs are also used in other areas of research such as simulation (Schruben, 1995) and automatic control (Libeaut, 1995). In this work ESGs are used for modeling system behavior, test generation, and robustness testing.

Test generation based on finite-state models has been an active area of research for many decades. Chow proposed the W-method for generating tests from finite-state models in the context of protocol testing (Chow, 1978). Chow's work was followed by work that generated variations of the W method, such as the Wp method (Fujiwara, 1991), new methods such as Unique Input-Output Sequences (Sabnani, 1988), Transition Tour (Aho, 1991), Distinguishing Sequences (Sarikaya, 1989), and empirical evaluations (Petrenko, 1996; Shehady and Siewiorek, 1997; Bochmann, 1994). Finite-state models have also been proposed as a means for the specification and analysis of system behavior (Parnas, 1969; Shaw, 1980; Raju and Shaw, 1994). More recently, researchers have proposed ways to generate tests from a variety of UML specifications such as statecharts and sequence diagrams. Two doctoral theses offer algorithms to generate tests from statechart specifications (Bogdanov, 2000; Burton, 2002). Other authors suggest coverage and mutation based adequacy heuristics to generate tests from statecharts (Fabbri, 1999; Offutt, 2003).

White and Almezen took a different approach to the problem of test generation using finite-state models (White and Almezen, 2000). Their work is in the context of generating tests for testing GUIs. Rather than use the traditional Mealy or Moore machines, they propose an alternate representation of user-responsibilities using the idea of Complete Interaction Sequences (CIS). A CIS is represented using a finite-state model where user actions, such as OPEN FILE and EDIT, label the states and the edges are unlabeled. Thus the expected behavior in response to an event is implicit and specified elsewhere in contrast to the traditional finite-state models that indicate explicitly the system response to an input as an output label on each transition. The entire system is modeled as a collection of CISs. An advantage of the CIS-based approach lies in its scalability and intuitiveness. Instead of creating a single composite finite-state model, multiple CISs, each representing a user responsibility, are created thereby simplifying the task of model construction and test generation.

Memon et al. (Memon et al., 2000) have proposed a novel approach for GUI testing. Their approach deploys methods from knowledge engineering to generate test cases, test oracles, etc., and to handle the test termination problem.

The approach presented here is different from the finite-state based approaches in that ESGs are based on a finite sequence of events, rather than states. It uses a unified view where user or environment actions as well as system responses are considered as events for the purpose of modeling, though they are separated into different sets during testing. ESGs themselves model finite-event processes. A collection of ESGs, and not a single ESG, models the behavior of an entire system. The approach, however, differs significantly from others in two key respects: modeling and test generation.

ESGs are a representation used to model the discrete interactions between an event-based system and its environment. Thus, while ESGs are excellent for modeling user-GUI interactions in a way similar to that done using CISs, they have been also used here to model the behavior of real-time systems, e.g., a railway-crossing system. ESGs allow a modeler to think in terms of system "events" instead of system "states." Our experience with designers and programmers in the real-time software industry suggests that ESGs are often easier to use for modeling the discrete behavior of a system than the traditional finite-state machine that requires the use of "states." The idea of inversion, or complementing, makes ESG based modeling distinct from other test generation approaches. ESGs and their complements allow modeling the desirable behavior of a system in the presence of both expected and unexpected inputs as events. The latter model allows for the quantification of the robustness of a system and hence raises the possibility of incorporating system robustness into its overall reliability. While the inversion of finite-state machines needs some theoretical skills, inversion of ESGs is intuitive and easily done by a test designer without recourse to automata theory.

Another distinctive feature of the ESG-based approach lies in its test generation algorithm. Most finite-state based test generation methods focus on some form of coverage, e.g., transition coverage (Aho, 1991; White, 2000; Offutt, 2003), or on coverage and state identification (Chow, 1978; Sabnani, 1988). The ESG-based test generation algorithm achieves complete coverage of ESGs (Belli, 2001; Marré and Bertolino, 2003) through the use of the Chinese Postman problem (Edmonds and Johnson, 1973; Kwan, 1962) for managing round trips. In addition, the algorithm also formalizes and generalizes the notion of pair-wise testing (Tai and Lie, 1992) by including the ability to generate tests that cover all possible n-tuples for some $n>1$. While the number of tests so generated can be impractically large, the algorithm can be applied selectively to individual ESGs with different values of n. This feature renders the algorithm *customizable* to the criticality needs of a system. ESGs that correspond to the most critical portions of a system can be tested more thoroughly using higher values of n. Note that higher values of n allow the generation of tests that enable testing a system for errors revealed only through specific sequences of inputs; such errors are known to be hard to find.

Another state-oriented group of approaches to test case generation and coverage assessment is based on model checking, e.g., the SCR (Software Cost Reduction) method, as described in (Gargantini and Heitmeyer, 1999). These approaches identify negative and positive scenarios to generate test cases automatically from formal requirements specifications. Thus they attempt to overcome the problem of testing that is not exhaustive, e.g., "black-box checking", which combines "black-box testing" and "model checking" (Peled, 2001).

Several approaches have been proposed to assess the robustness of software-based systems. Fetzer and Xiao (Fetzer, 2002) have proposed techniques for increasing the robustness of C libraries using wrappers. Huns and Holderfield (Huns, 2002) equate robustness to a lack of software crashes and suggest redundancy and appropriate granularity as a way to achieve it. Kropp et al. have proposed an automated method, the Ballista approach, for testing the robustness of software (Kropp, 1998). Ballista employs random test generation. The proposed ESG approach differs from the approaches cited here in that it (a) allows the modeling of incorrect behavior that is often the cause of lack of robustness of software systems and (b) provides an algorithmic approach to test generation for testing a software-based system for robustness.

3. Event Sequence Graphs

In this section we introduce event sequence graphs (ESGs) as a behavior-modeling device; both the desirable and undesirable behavior are modeled using ESGs. Any software-based system can be viewed as interacting with its environment through stimuli-response pairs. In this context, the environment could be one or more human users, a set of service seekers, or any combination thereof, and we use the terms "user" and "environment" interchangeably.

An *Event Sequence Graph (ESG)* is a device to model a subset of the interactions between a system and its user. The complete set of interactions is captured in terms of a set of ESGs, where each ESG represents a possibly infinite set of event sequences. An event, an externally observable phenomenon, can be a user stimulus or a system response, punctuating different stages of the system activity.

Formally, an ESG is a labeled graph represented as a triple $ESG = (\alpha, E)$ where α is a finite set of labeled nodes (vertices) and E: $\alpha \rightarrow \alpha$, a relation, possibly empty, on α. The elements of E represent directed arcs (edges) between the nodes in α. As a convention, two distinct vertices are identified as *entry* and *exit* nodes and are shown respectively by an incoming arc with no source and an outgoing arc with no destination.

Elements of α are known as *events*. The set α is partitioned into two subsets α_{env} and α_{sys} such that

$$\alpha = \alpha_{env} \cup \alpha_{sys}, \qquad \alpha_{env} \cap \alpha_{sys} = \phi, \tag{1}$$

where α_{env} is the set of *environmental events* (i.e., user inputs) and α_{sys} a set of *system responses*. The distinction between the sets α_{env} and α_{sys} is important because the events in the latter are controllable within the system, whereas the events in the former are assumed to be not subject to such control.

We refer to a node in α by its label. Given two nodes a and b in α, a directed arc from a to b indicates that event b can follow event a. A simple ESG is shown below.

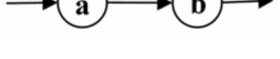


Fig. 1. An ESG with two nodes labeled *a* and *b*

The complement of an ESG, denoted by $\overline{ESG}$, includes all the edges not included in the ESG, but with no new entry or exit edges. The complement of the ESG in *Fig. 1* appears in *Fig.2*.

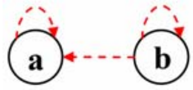


Fig. 2. $\overline{ESG}$ – Complement of the ESG given in *Fig. 1*

Finally, the *completed ESG*, referred to as *CESG,* is constructed as the superposition of the ESG and its complement $\overline{ESG}$. For example, superposition of the ESG in *Fig. 1* and its complement in *Fig. 2* leads to a CESG shown in *Fig. 3*.

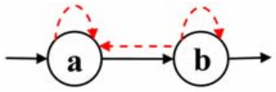


Fig. 3. CESG – Completed ESG of the ESG given in *Fig. 1*

Interaction patterns implicit in an ESG, $\overline{ESG}$, and *CESG*, can also be expressed in terms of a regular expression (Salomaa, 1969; Shaw, 1980). For example, the following regular expression *r* captures all interaction patterns implicit in the CESG in *Fig. 3*.

$$r = (a^+b^+)^+ \qquad \textbf{(2)}$$

Expression (2) indicates that a sequence of one or more events *a* can be followed by a sequence of one or more events *b* and, furthermore, this pattern can recur one or more times. Note that Kleene's star operation "*", which is not used in this example, indicates an arbitrary number of occurrences, including the *empty* sequence. Note also that a CESG and the corresponding regular expression capture, respectively, all valid and invalid sequences of events.

Let *R* denote the regular set denoted by the regular expression *r*. Given a system *M* and an ESG M_e that models a set of interactions between the user and *M*, we refer to the corresponding regular expression as $r(M_e)$ and the regular set as $R(M_e)$. Thus $R(M_e)$ denotes a possibly infinite set of strings, or event sequences in the present context, over the alphabet α. Often, it is a finite set of ESGs that model all interactions of concern with *M*. We shall use *R(M)* to denote the set of all interactions modeled by the ESGs in this set. It is easy to obtain *R(M)* as $R(M_{e1}) \cup R(M_{e2}) \cup ... \cup R(M_{en})$ given the *n* ESGs $M_{e1}, M_{e2}, ...M_{en}$ that model the behavior of *M*.

ESGs are comparable to the Myhill graphs (Myhill, 1957), which are also adopted as computation schemes (Ianow, 1958), or as *syntax diagrams*, e.g., as used in (Jensen and Wirth, 1974) to define the syntax of Pascal. The difference between the Myhill graphs and the ESGs as introduced here is that the symbols, which label the nodes of an ESG, are interpreted not merely as symbols or meta-symbols of a language, but as operations on an event set (see also *Event Sequences* (Korel, 1996)). A flexible visualization at different abstraction levels is given through *view graphs* (Gossens, 2005).

3.1. Modeling Functions and Malfunctions

We use the term *system function,* or simply *function,* to refer to the correct behavior of the SUT while the term *malfunction,* or *dysfunction*, refers to its incorrect behavior. Using the event terminology above, functions and malfunctions can be represented as regular expressions over the set α.

For a system *M,* event sequences over α that belong to *R(M)* denote system functions, while others denote malfunctions. Let *F* denote the set of *system functions* and *D* the set of system *malfunctions*. We then have

$$F \subseteq R(M) \text{ and } D \subseteq \overline{R(M)} \qquad \textbf{(3)}$$

To test a system, one generally produces meaningful test *inputs* and complies a list of corresponding expected system *outputs*. Accordingly, a *test* represents the execution of the SUT and comparison of the outcome with the expected. When the test results are in accordance with the user's expectations, the test *succeeds* otherwise it *fails*.

Some nodes in an ESG represent environmental events, e.g., user inputs lead to expected system responses, which are also considered as events. Thus, each edge of the ESG represents a *legal event pair,* or simply, an *event pair (EP)*. For example, *ab* in *Fig. 1* is an event pair. A sequence of *n* consecutive edges is an *event sequence (ES) of the length n+1.*

A *complete ES (CES)* starts at the entry of the ESG and ends at its exit, i.e., it represents a *walk* through the ESG. The set of the CESs specifies the system function *F* as introduced in (3). Alternately viewed, the CESs constitute legal words of the regular set defined by an ESG.

Analogous to the notion of EP, *faulty (or illegal) event pairs (FEP)* are introduced as the edges of the corresponding $\overline{ESG}$, e.g., *aa, ba, bb* in *Fig. 3*. Further, an EP of the ESG can be extended to a *faulty*, or an *illegal*, *event triple (FETr)* by adding a subsequent FEP (if there exists one) to this EP, e.g., *ab* and *bb* of *Fig. 3*, resulting in *abb*. Thus a FETr consists of three consecutive nodes in an ESG where the last two nodes constitute an FEP. In general, a *faulty event sequence (FES)* of the length n consists of *n-1* events that form a (legal) ES of length *n-2* and of two events at the end that form an FEP.

Given an ESG *e,* faulty CESs (FCESs) can be constructed systematically using FEPs as follows.

- An FEP that starts at the entry of *e* is also an FCES. **(4)**
- An FEP *f* that does not start at the entry of *e* is not executable and is extended by adding suitable prefixes. Each ES that starts at the entry of *e* and ends at the first symbol of *f* is prefixed to *f* and the resulting sequence becomes an FCES. Such prefixes are referred to as *starters*.

Note that the attribute "complete" in FCES expresses only the fact that an FEP might have been "completed" by means of an ES as a prefix to make it executable (otherwise it is *not* complete, i.e., not executable). Thus, an FCES is an FES that starts at the entry node but fails to reach the exit node. For a given SUT, we refer to the set of FCES as the set *V.*

3.2. Risks and Risk Ordering

Malfunctions of a system are often related to its state. However, since the representation based on ESGs is void of any explicit notion of state, it is necessary to refer to states indirectly in terms of the elements of $R(M)$, which as explained earlier, are event sequences beginning at the entry node. Thus, a string in $s \in R(M)$, s may also be treated as a notation for the state reached by *M* upon the execution of the events in s.

In embedded systems, such as a pacemaker or a railway-crossing controller, an event sequence s may lead the system to a state that has some form of risk associated with it. Though we do not concern ourselves with the actual quantification of risk, we need an ordering relation based on risk for the states of *M*. As shown in Section 6, this offers us a systematic selective approach to test generation.

A *risk ordering relation* □ is defined as

□ = {(s1, s2)| s1, s2∈$R(M)$ and the risk level associated with state s1 is less than that associated with state s2.} **(5)**

The risk ordering function above is analogous to that used in (Nissanke and Dammag, 2002). In this context, *risk level* of a state quantifies the *"degree of the undesirability"* of an event sequence from the perspective of some critical system attribute, which could be, for example, safety. An analogous interpretation of □ can be found for other system attributes.

The risk ordering relation □ is intended as a guide to determining an appropriate response to any threats detected. Such responses are specified in terms of a *defense matrix DM* that utilizes the risk ordering relation to revert the system state from its current one to a less, or the least, risky state. *DM, and the associated constraint,* are defined as follows.

$$DM \in R(M) \times D \rightarrow R(M) \qquad \textbf{(6)}$$

$$\forall\, s_1, s_2, v \bullet (s_1,v) \in \text{dom } DM \wedge DM(s_1,v) = s_2 \Rightarrow s_2 \quad s_1 \qquad \textbf{(7)}$$

The above expresses the requirement that, should one encounter the malfunction d in any given state s_1, the system must be brought to a state s_2, which is of a lower risk level than s_1. A defense action, which is an appropriately enforced sequence of events, is used to bring the system into a less risky state. An exception handler executes a defense action. The actual definition of the defense matrix and the appropriate set X of exception handlers is the responsibility of a domain expert specializing in the risks posed by a given malfunction. Relying on this risk based interpretation of state in terms of event sequences, if x is a defense action appropriate for the scenario implicit in (5), then s_1 $x = s_2$.

A specific benefit of risk ordering in the framework introduced here is that it allows a systematic approach to the selection of test cases by focusing on one or more particular vulnerability attribute.

3.3. Quantification of Robustness

Robustness of a software system is defined as the ability of the system to behave acceptably in the presence of unexpected inputs (Huhns, 2002). We prefer to treat robustness as the ability of a system to handle exceptional or faulty inputs. Thus, while there is an expected set of inputs, its complement is a faulty set of inputs. The ability of a system to handle acceptably such exceptional inputs is a measure of its robustness. A set of ESGs that models a SUT behavior defines the set of expected event sequences. The complement of each ESG in this set, taken together, defines the set of unexpected input sequences. We define robustness with respect to precisely this set of unexpected input sequences.

Several approaches have been proposed to assess the robustness of a system. The Ballista approach (Kropp, 1998) is an elegant way to assess the robustness of a software system by the generation of special values and random inputs. Here we propose an alternate ESG based approach to testing a software system for robustness. This approach allows the quantification of robustness with respect to a universe of erroneous inputs.

As mentioned above, in Section 3, the complement of an ESG defines a subset of all possible erroneous or faulty, event sequences. A set of erroneous inputs is obtained for the SUT by complementing each ESG in the set that models the behavior of the SUT. We explain in Section 4.3.2 how these faulty sequences are used to generate test inputs that test the exception handling ability of the SUT. Given that there are n tests, each containing a faulty sequence, and that in m, $m \leq n$, of these tests the SUT behaves acceptably, the robustness of the SUT is estimated to be the ratio m/n.

The robustness measure proposed above is significantly different from the one obtained using the Ballista approach. While the Ballista approach uses special and random values of SUT input variables to assess robustness, the ESG approach uses tests based on faulty event sequences as a way to assess the same. A comparison of the two approaches is not within the scope of this work.

3.4. Fault Model

As ESGs are constructed according to the user expectation concerning the system behavior, any CES is supposed to successfully run the system. Thus, a CES can be used as a test case, and the test fails if this CES starting at the entry node

- cannot reach the exit node due to a failure, e.g., system crash (*sequencing fault*), or
- reaches the exit node, but does not deliver the expected operation result (*functional fault*).

Accordingly, a CFES is supposed to cause a failure, or if there is an exception handling mechanism, an error message about the impairment of the events; otherwise a sequential fault occurs. A sequencing fault can also occur in the starter portion, i.e., in the ES as the prefix of the FCES. CFES-based functional faults do not make any sense as they are supposed to exclude the expected behavior of the system.

This fault model is very simple and makes clear how the oracle problem is handled: A CES-based test case is supposed to succeed a test whereby a FCES-based one to fail the test.

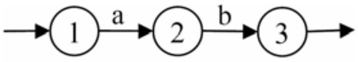
In spite of its simplicity, the fault model is sufficiently powerful to guarantee revealing all sequencing faults, provided that ESs and FCESs to be covered are sufficiently long (Sections 4.3.1 and 4.3.2). The approach cannot, however, guarantee to detect all functional faults, because in case a test succeeds, the user must validate that the expected result has been obtained.

3.5. Handling Other Features

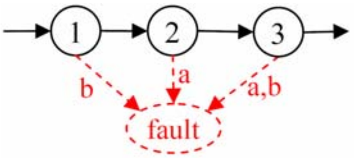
3.5.1. States and Outputs

Traditional finite-state automata (FSA) consist of states and transitions labeled by inputs, and in the case of a Mealy machine, also outputs. While an ESG is a finite, memoryless, device, in the sense that it consists of a finite set of nodes and vertices, the transitions are unlabeled. Merging the states and inputs/outputs of the FSA to derive the corresponding ESG considerably simplifies the fault modeling.

As an example, the ESG of *Fig. 1* is represented as an FSA (*Fig. 4*, *(a)*) which then is completed by a fault state (*Fig. 4*, *(b)*). *Fig. 3* is then compared with *Fig. 4*, *(b)* in order to illustrate the fault modeling features in FSAs.



(a) FSA which is equivalent to the ESG of *Fig. 1*



(b) Completed FSA

Fig. 4. Completed FSA of *Fig. 1*, leading to a total of 6 edges

If the underlying ESG has n vertices, the corresponding CESG has at most n^2 edges that connect each of the n vertices with every other vertex (West, 1996). The ESG in *Fig. 1* has two events, leading to a total of 4 edges ($2^2 = 4$) of the CESG in *Fig. 3*, without counting the entry and exit edges. Assuming that the corresponding FSA in *Fig. 4(a)* has three states and an input alphabet of two symbols, *a* and *b*, the corresponding, *completed FSA* (*CFSA*) is given in *Fig. 4* (b) with an extra state “fault.” For the sake of simplicity, edges are allowed to be associated with multiple inputs, e.g., with *a, b*. Evidently, a CFSA with n states and an input alphabet of the cardinality m has $m \cdot n$ edges (without counting the entry and exit edges). Thus, the example CFSA in *Fig. 4* has a total of 6 edges (2·3 = 6); with the edge labeled with two inputs counted as a double edge.

3.5.2. Multiple Start and Final Events

In the definition above, ESGs have a single start node at the entry and a single final node at the exit. In cases where multiple start nodes and/or multiple final nodes are needed, additional (pseudo) single entry node, denoted by *[*, and/or a single pseudo exit node, denoted by *]*, are introduced and connected to the corresponding multiple nodes required at the entry and/or at the exit. In the ESG of *Fig. 5*, *a* is the start node that is connected to the pseudo entry node *[*, and *a* (the second occurrence) and *c* are the final nodes that are connected to the pseudo exit node *]*. The same notation is applicable to ESGs with single entry and exit nodes, though in this case pseudo nodes, hence the square bracket notation, may be dropped without causing any confusion. Any direct transition from *[* to *]*, if added, indicates that the ESG can generate an empty event sequence.

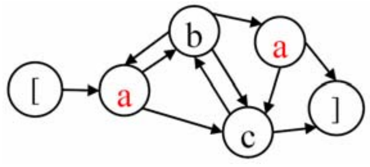


Fig. 5. Pseudo start/final nodes (*[,],* respectively) and interaction ambiguities (caused by the double occurrence of *a*)

3.5.3. Handling Context Sensitivity

When using ESGs to model an application, e.g., a graphical user interface, there is often a need for using the same command, or the same icon, for similar operations in different contexts or in different hierarchical levels of the application. An example is the operation `delete` used for deleting a symbol, a record, or even a file. In such cases, the system usually carries out the proper action using the context information. The approach introduced, however, eliminates the need for being explicit about the hierarchy information in abstracting the real system into an ESG model.

As an example, *Fig. 5* depicts an ESG that has two different nodes with the same label *a* and therefore, can be initiated or triggered by the same input *a*. While constructing the EPs and FEPs, and accordingly the CESs and FCESs, one needs to differentiate between the node *a* that leads to *b* or *c*, and the node *a* that can be reached via *b* and leads only to *c*. This ambiguity can be resolved simply by indexing, for example, a_1 identifying the first appearance of *a*, and a_2 identifying the latter one. This indexing implies the syntactical, or contextual, position and can help with the reconstruction of different hierarchical levels that have been "flattened" in the course of modeling.

3.5.4. Extension of the Fault Model

Based on past work related to fault modeling (Eggers, 1984; Belli, 1991; Fabbri, 1994; Delamaro, 2001; DeMillo, 1978, 1991) and denoting inputs, outputs, states, or transitions as *elements*, we obtain the following fault model.

- *Omission error* (*o-error*) – an element has been omitted.
- *Insertion error* (*i-error*) – an element has been inserted
- *Corruption error* (*c-error*) – an element has been corrupted.

Note that a *c*-error can be represented by an *o*-error followed immediately by an *i*-error, with a different element being inserted for the omitted element.

When applied to the elements of a Mealy automaton, these hypotheses are capable of delivering test cases to detect a variety of defects, e.g., whether an edge is missing as a result of a defect of the next state function, or if an output is missing or corrupted, since the output function does not work properly, etc. (Gill, 1962). The hypotheses can be extended from single errors to multiple (*n*) errors (Eggers, 1984)**:**

- o^n*-errors* – *n* elements have been omitted.
- i^n*-errors* – *n* elements have been inserted.
- c^n*-errors* – *n* elements have been corrupted.

Finally, to represent arbitrary types of faults within the context of a finite-state model, an appropriate combination of these hypotheses is necessary, e.g., "a transition is forgotten, or inserted, or two transitions have been interchanged" can be represented by $o + i^2 + c$, where "+" represents the logical operator for "(exclusive) or". The described fault model can generate many classification schemes for coverage, based on, for example, (Chillarege, 1996; Offutt et al., 2003; Marré, 2003).

This extension can also be applied to ESG-based modeling, and enables, in turn, a precise assignment of severity levels to undesirable events in accordance with experience and judgment of the tester (see Section 5.8).

4. Test Generation

We now focus on test generation from an ESG model. The method described in this section uses ESGs and their complements as inputs and generates a test set that is complete with respect to a model-based coverage criterion.

4.1. Objectives

As already mentioned when discussing the fault model (Section 3.4), a CES, by definition, is expected to lead a SUT to a desirable state, and hence it may be viewed as a test input against which the SUT is expected to produce a correct output. The generation of CESs is one of the objectives of the test generation procedure described. The other objective is to generate FCESs from the complement of ESGs that together model the whole system behavior – both the desirable and the undesirable parts. Upon the input of an FCES, the SUT is expected to transfer itself temporarily into a faulty state and might invoke a fault detection/correction procedure, provided that an appropriate *exception handling* mechanism has been implemented (Goodenough, 1975; Randell, 1978). Thus, while CESs are used to test for the correct behavior of an SUT, the FCESs are used to check for the correctness of exception handling.

We seek a test generation algorithm that, given an ESG and the corresponding CESG, generates tests that satisfy the following coverage criteria.

- Cover all event pairs in the ESG, and **(8)**
- Cover all faulty event pairs of the CESG.

Note that a test set that satisfies the first of the two criteria above consists of CESs while the one that satisfies the second consists of FCESs. Thus all transitions in the ESG and CESG will necessarily be covered by the CESs and FCESs, respectively (Belli, 2001; Belli and Dreyer, 1997). However, this criterion is more powerful than the transition or node coverage criterion (Offutt, 2003) as, in addition, it requires the coverage of pairs of adjacent nodes in the ESG and its corresponding CESG. We will show later in Section 4.3 how this pair-wise coverage can be generalized to n-tuple, n>2, coverage and the cost and benefits of such an extension.

It is obvious that there exist a large number of solutions to the test generation problem as stated above. For example, when an ESG has a loop, one could obtain a long chain of events that constitute a CES. This observation leads us to impose the following additional constraints on the test generation process:

- The sum of the lengths of the generated CESs should be minimal. **(9)**
- (b) The sum of the lengths of the generated FCESs, should be minimal.

The constraint on lengths of the tests generated allow for a reduction in the cost of test execution. One might argue that minimizing test length might have an adverse effect on fault detection. While this is true in general, our experiments show that the effect is minimal. Further, the coverage criteria can be made more powerful by increasing the value of n in the n-tuple coverage to be obtained thereby further reducing any negative effect of reducing the length of tests on the fault detection effectiveness.

The set of CESs that satisfy (7a) and (8a) for a given ESG is referred to as the minimal spanning set for the coverage of event sequences of ESG (MSCES). An MSCES is a complete and minimal set of test cases aimed at exercising all event-sequences of a given length and related to the desirable behavior of the SUT. Similarly, the set of FCESs that satisfy (7b) and (8b) is referred to as the minimal spanning set for the coverage of faulty event sequences (MSCFES). A MSFCES is a complete and minimal set of test cases aimed at exercising the SUT against faulty input sequences that test the exception handling behavior of the SUT.

4.2. Test Process

Once the tests have been constructed, they are input to the SUT. In case a CES is input, we check if the system behaves as expected on a correct input. In the case an FCES is input, we check if the system is able to recover from faulty inputs. The lack of a system's ability to respond as desired to an FCES is considered as a lack of robustness. When an FCES is input, an undesirable behavior might occur, for example, because an exception handler is missing or incorrectly implemented.

A major problem is the determination of the correct (i.e., desirable) and faulty (i.e., undesirable) behavior, also known as the oracle problem (Binder, 2000; Hamlet, 1994). The present approach handles the oracle problem effectively by embedding the expecting behavior within the CES itself. Recall that both types of events, from the environment and responses generated by the system, are a part of a CES.

4.3. Test Generation and Execution Algorithm

In this section we sketch the algorithms used for the generation of CESs and FCESs given an ESG and its complement. Algorithms presented here are extensions of the well-known algorithm for solving the Chinese Postman Problem (CPP) (Edmonds, 1973).

A solution to the CPP is a minimum length closed walk that covers each edge of the given graph at least once. A solution to this problem for a given ESG satisfies the first constraint in (9). However, such a solution might fail to satisfy the first constraint, and its generalization to triples, quadruples, etc., in (8) which requires that all event pairs be also covered. It is the satisfaction of (8) and its generalization that requires an extension to the algorithm for solving the CPP. A similar extension is also needed for generating FCESs from CESG.

Algorithm 1. Test Generation and Execution Algorithm

```
Input: an ESG with
n:= number of the functional units (modules) of the system that fulfill a well-
     defined task
length:= required length of the event sequences to be covered
   FOR unit 1 TO n DO
   BEGIN
      Generate appropriate  ESG  and  $\overline{ESG}$
      FOR k:=2 TO length DO
      BEGIN
         Cover all ESs of length k by means of CESs subject to
         minimizing the number and total length of the CESs
      END
      Cover all FEPs of unit  by means of FCESs subject to
      minimizing the total length of the FCESs
   END
   Apply the test set given by the selected CESs and FCESs to the SUT.
   Observe the system output to determine whether the system response is in
   compliance with the expectation.
```

4.3.1. Minimal Spanning Set for the Coverage of ESs

As mentioned earlier in Section 3, 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 (West, 1996), MSCESs can be constructed as follows:

- Check whether an ideal walk exists.
- If not, check whether a complete walk exists and construct a minimal one if it does.
- 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.

Construction of MSCES

The MSCES problem introduced here is expected to have a lower degree of 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 and that make the test process scalable.

An algorithm described in (Thimbleby, 2003) 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 *Fig. 6*. Addition of a backward edge, indicated as a dashed arrow from the exit to the entry, transforms the ESG in *Fig. 6 (a)* to a strongly connected graph in *Fig. 6 (b).*

The labels of the vertices in *Fig. 6 (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 the all-shortest-paths and solving the optimization problem. The problem can then be transformed into the construction of an Euler tour for this graph (West, 1996). This tour may contain the backward edge multiple times, indicating the number of walks.

For the example given in *Fig. 6 (a)*, the minimal tour (based on *Fig. 6 (b)*) and the minimal set of the legal walks (i.e., CESs) covering the EPs is given by:

$$\text{Minimal Tour} = (abcbdcaca); \quad \text{MSCES} = \{ abcbdc, ac\} \tag{10}$$

Note that no entire walks exist. Therefore, an ideal walk cannot be constructed. Furthermore, the MSCES can be constructed in time $O(|\alpha|^3)$ (West, 1996). See Algorithm 2 for details.

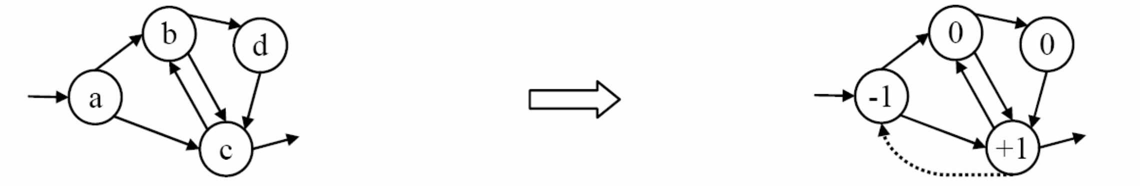


Fig. 6 (a). An example ESG

(b). Transferring walks into tours and balancing the nodes

Algorithm 2: Generation of MSCES

Input: ESG=(V, E); ε=[, γ=]
Output: MSCES

```
add_arc(ESG, (γ, ε));
sets A, B, M, MSCES = ∅;
FOR all nodes v∈V DO
  IF (diff(v) > 0) THEN
    A = A ∪ {$v_i$ | i∈{1,..,diff(v)}};
  IF (diff(v) < 0) THEN
    B = B ∪ {$v_i$ | i∈{1,..,diff(v)}};
m = |A| = |B|;
D[1 .. m][1 .. m];
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)
    MSCES = MSCES ∪
        getPartialList(EulerTourList, start, i);
    start = i + 1;
RETURN MSCES;
```

Theorem 1: MSCES can be constructed in time $O(|V|^3)$.
Proof (sketched; see also H. Thimbleby, 2003): The shortest paths from one node to all other ones can be determined by the depth-first-search in $O(|E|+|V|)$, as ESG under consideration is a unweighted graph. Furthermore, because of $|E|>>|V|+1$, the complexity can be approximated to

O(|E|). Following the shortest paths of all nodes to all other ones can be determined by $O(|E|*|V|)$. The Hungarian Algorithm that solves the assignment problem has the complexity $O(|V|^3)$ and the algorithm next to determine the Euler-Tour has the complexity $O(|E|*|V|)$. Thus, the total complexity is determined by $O(|V|^3)$.

4.3.2. Minimal Spanning Set for the Coverage of FESs

In comparison to the interpretation of the CESs as legal walks, by definition illegal walks are realized by FCESs that never reach the exit node. An illegal walk is *minimal* if the length of its starter cannot be further reduced.

Assuming that an ESG has n vertices and d edges as EPs, then exactly $u = n^2 - d$ edges are the FEPs. Thus, at most u FCESs of minimal length, i.e., of length 2, are available. These FCESs emerge when the node (nodes) following the entry node is (are) followed immediately by a faulty input. Accordingly, the maximal length of an FCES can be n; these are subsequences of CESs with their last event being replaced by an FEP. Therefore, the number of FCESs is determined precisely by the number of FEPs. An FEP that represents an FCES is of a constant length of 2 and therefore 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 (Dijkstra, 1959). Below is the minimal set of the illegal walks for the graph in *Fig. 6 (a)*.

$$\{\ aa,\ ad,\ aba,\ abb,\ aca,\ acc,\ acd,\ abdb,\ abda\ \} \quad \textbf{(11)}$$

While constructing the MSCESs, we take into account the ESs that are used to form starters to construct MSFCESs. The ESs used as starters need not be covered by additional CESs. This can help save costs if the test budget is limited, as is often the case in practice.

The determination and specification of the CESs and FCES should ideally be carried out during the definition of the user requirements, often long before the system is implemented. They are then a part of the system test specification. Certainly, CESs and FCESs can also be produced incrementally at any later time, even during testing.

4.3.3. Generating Event Sequences with Length > 2

A phenomenon in testing interactive systems that most testers seem to be familiar with is that faults can be frequently detected and reproduced only in some context. This makes a test sequence of a length>2 necessary since repetitive occurrences of some subsequences are needed to cause an error to occur/recur.

Consider the following scenario: Based on the ESG given in *Fig. 7*, the tester assumes that the EP given by ***BC*** always reveals a fault, no matter if executed within *[A**BC**]*, *[ABA**BC**]*, or *[ABDC**BC**]*; i.e., the test cases containing *BC* always detect the fault in any context. In this case, the fault is said to be a *static* one, as it can be detected without a context. Furthermore, the same scenario (so the assumption) demonstrates that the EP ***BA*** reveals another fault, but only in the context of *[ABC**BA**C]*, and never within *[A**BA**C]*, or *[A**BA**CBDC],* etc. In this case the fault is said to be a *dynamic* one.

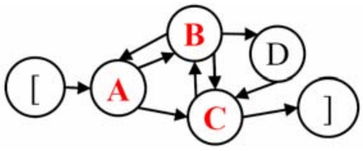


Fig. 7. Static faults vs. dynamic faults (discussed events are boldface)

Such observations clearly indicate that the test process must be applied to longer ESs than 2 (EPs).

Therefore, an ESG can be transformed into a graph in which the nodes can be used to generate test cases of length > 2, in the same way that the nodes of the original ESG are used to generate EPs and to determine the appropriate MSCES.

Fig. 8 illustrates the generation of ESs of length=3. In this example adjacent nodes of the extended ESG are concatenated, e.g., AB is connected with *BD*, leading to *ABBD*. The shared event, i.e., *B*, occurs only once producing *ABD* as an ES of length=3. In case ESs of length=4 are to be generated, the extended graph must be extended another time using the same algorithm.

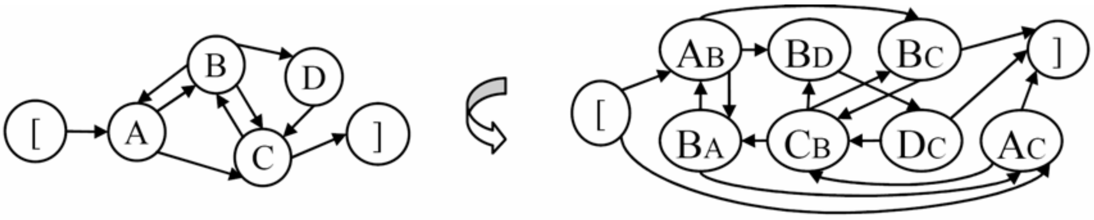


Fig. 8. Extending the ESG for covering ESs of length=3

The common procedure embodying this approach is given in Algorithm 3.

Algorithm 3: Generating ESs and FESs with length > 2

Input: *ESG=(V, E)*; *ε* = [, *γ*=], *ESG'=(V', E')* with *V'*=∅ , *ε'*= [, *γ'*=];
Output: *ESG'=(V', E')*, *ε'*= [, *γ'*=];

```
FOR EACH (i,j)∈E with (i<>ε) AND (j<>γ) DO
  addNode(ESG',(ES(ESG,i) ⊕ ω(ES(ESG,j)));
  removeArc(ESG,(i,j));
FOR EACH i∈V' with (i<>ε') AND (i<>γ') DO
  FOR EACH j∈V' with (j<>ε') AND (j<>γ') DO
    IF(ES(ESG',i) ⊕ ω (ES(ESG',j)) =
                    α (ES(ESG',i)) ⊕ (ES(ESG',j)) THEN
      addArc(ESG',(i,j));
  FOR EACH (k,l)∈E with k=ε DO
    IF(ES(ESG',i) = ES(ESG,l) ⊕ω (ES(ESG',i)) THEN
      addArc(ESG',(ε',i));
  FOR EACH (k,l)∈E with l=γ DO
    IF(ES(ESG',i) = α (ES(ESG',i)) ⊕ ES(ESG,k) THEN
      addArc(ESG',(i,γ'));
```

```
RETURN ESG';
```

In Algorithm 3, `ES(ESG,i)` represents the identifier, e. g., *AB*, of the node *i* of the *ESG*. This identifier can be concatenated with another identifier `ES(ESG,j)` of the node *j*, e.g., *CD*. This is represented by *AB* ⊕ *CD*, or `ES(ESG,i)` ⊕ `ES(ESG,j)`, resulting in the new identifier *ABCD*. Note that the identifiers of the newly generated nodes to extend the ESG are constructed using the identifiers of the existing nodes. The function `addNode()` inserts a new ES of length *k*. Following this step, a node *u* is connected with a node *v* if the last *n-1* events that are used in the identifier of *u* are the same as the first *n-1* events that are included in the identifier of *v*. The function `addArc()` inserts an arc, connecting *u* with *v* in the ESG. The pseudo nodes [,] are connected with all the extensions of the nodes with which they were connected before the extension. In order to avoid traversing the entire matrix, arcs which are already considered are to be removed by the function `removeArc()`.

Apparently, this Algorithm 3 has a complexity of $O(|V|^2)$ because of the nested FOR-loops to determine the arcs in the *ESG'*. Another algorithm to generate FESs of length > 2 is not necessary because such faulty sequences will be constructed through the concatenation of the appropriate starters with the FEPs. Algorithm 2 can be applied to the outcome of Algorithm 3, i.e., to the extended ESG, to determine the MSCES for *l(ES)* > 2.

5. Case Study 1: The RealJukebox

We now present a case study to determine the effectiveness of the tests generated using the algorithms mentioned in the previous section. The study was conducted using the RealJukebox of the RealNetworks application available in the public domain. We did not have access to the source code and any specification of the application, other than its user manual. Hence all ESGs required for test generation of were derived from the application GUI.

5.1. Objectives

The objective of this empirical study was to investigate the effect of varying

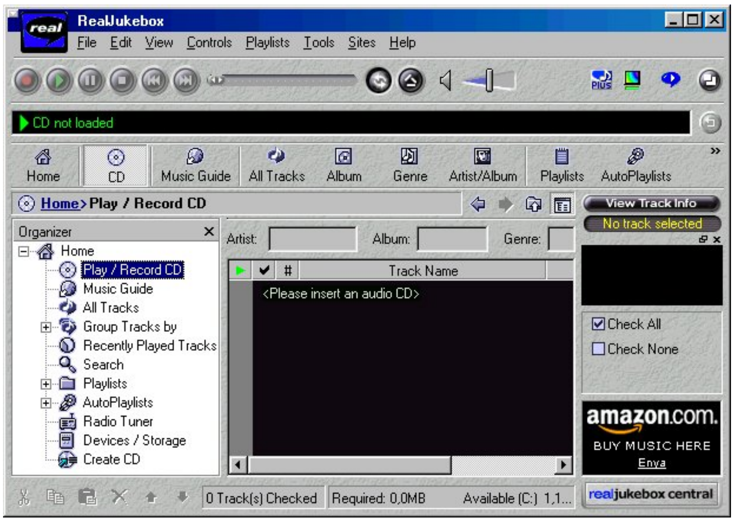
- event-tuple coverage, i.e., length of the ES to be covered, subsequently referred to as *n-tuple coverage*, and
- the number of the event sequences

on the fault detection effectiveness of the CESs and FCESs using the algorithms sketched in Section 4.3 (see Basili, 1986; Wohlin., 2001).

The value of *n* is considered as a contributor to the cost of the test process; the larger the value of *n* the more costly the test process in terms of the human effort spent in administering the test. We studied the fault detection effectiveness of the generated test set for n=2, 3, and 4, that correspond to, respectively, pair-wise, triple, and quadruple coverage.

5.2. System Description and Model

RealJukebox (RJB) is a personal music management system to build, manage, and play individual digital music library on a personal computer. *Fig. 9* is a snapshot of the RJB interface showing the main menu. At the top level, the GUI has a pull-down menu with the options *File, Edit*, etc., to invoke operations. These options have further sub-options, and so on. There are additional window components allowing navigation through the entries of the menu and sub-menus, creating many combinations and, accordingly, many applications.



CD not loaded

<Please insert an audio CD>

0 Track(s) Checked | Required: 0,0MB | Available (C:) 1,1...

Fig. 9. Example of a GUI (RealJukebox of RealTime)

In the course of the present case study, a set of ESGs was determined for the RJB. This task was performed manually by studying the online help function, the user manual, and the GUI of the RJB and identifying distinct functionalities from a user's point of view. The complete set of ESGs is found in Appendix A. As an example, the ESG in *Fig. 10* represents the top-level GUI to produce the desired interaction "`Play and Record a CD or Track`" via the main menu in *Fig. 9*. The user can load a CD, select a track and play it. One can then change the mode, replay the track, or remove the CD, load another CD, etc. *Fig. 10* illustrates all sequences of user-system interactions to realize the likely operations that the user might launch when using the system.

Each of the correct interactions, denoted by the nodes in *Fig. 10*, defines a system sub-function that must be refined and represented in a corresponding sub-ESG, as done in Section 5.7 for the node `P` (`Play track`) in *Fig. 14*.

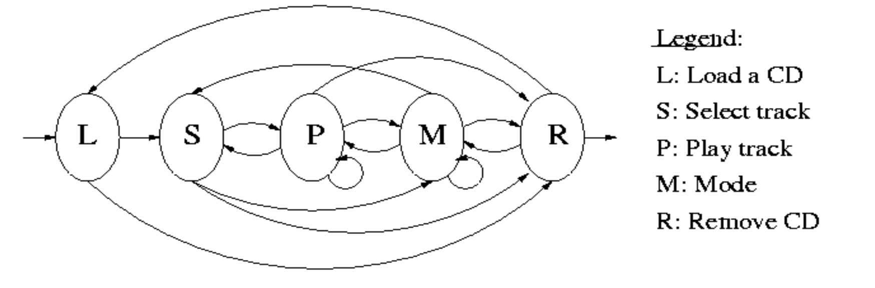


Fig. 10. The system function `Play and Record a CD or Track` represented as an ESG

The derivation of ESGs required some experience in designing GUIs and an understanding of how they function. As is common in modeling processes, the interactions that seem most relevant in the diagram are selected and named so as to reflect the user's perspective. We are not aware of any algorithmic ways to generate ESGs automatically from the GUI.

5.3. Test Representation

The nodes of an ESG are interpreted as operations on identifiable objects that can be controlled/perceived by input/output devices, i.e., elements of WIMPs (Windows, Icons, Menus, and Pointers). Thus an event can be a user input (an element of the set α_{env}, see (1)), or a system response (an element of the set α_{sys}), leading or triggering interactively to a succession of user inputs and system outputs. Accordingly, a chain of edges from one vertex of an ESG to another is realized by sequences of the form below.

initial user input(s)→ (interim) system response(s) → (interim) user input(s) → … → (final) system response **(12)**

(12) defines an ES as introduced in Section 3. An ES may consist of no interim system responses but only user inputs and a final system response as, for example, in *Fig. 10*. Note that our event sequences are similar to those used by White et al. (White and Almezen, 2000).

Given the ESG in *Fig. 10*, test generation begins with an analysis of the system function `Play and Record a CD or Track.` This analysis leads to the following set of EPs.

LS, *LR*, *SP*, *SM*, *SR*, *PS*, *PP*, *PR*, *PM*, *MP*, *MS*, *MM*, *MR*, *RL*, *RM* **(13)**

In the next step we generate CESs. As explained in (see Section 3.1), CES is a walk obtained by extending the EPs by appropriate prefixes and/or suffixes. The list (14) below gives the CESs as test inputs.

LSR, *LR*, *LSPR*, *LSMR*, *LSR*, *LSPSR*, *LSPPR*, *LSPR*, *LSPMR*, *LSMPR*, *LSMSR*, *LSMMR*, *LSMR*, *LRLR*, *LRMR* **(14)**

Some CESs, e.g., *LSR*, occur more than once in (14). This is because *LSR* can be obtained by adding the suffix *R* to the event pair *LS* as well as by adding a prefix *L* to the event pair *SR*. Elimination of this redundancy leads to (15).

LSR, *LR*, *LSPR*, *LSMR*, *LSPSR*, *LSPPR*, *LSPMR*, *LSMPR*, *LSMSR*, *LSMMR*, *LRLR*, *LRMR* **(15)**

The set of the CESs given in (15) ensures that all EPs are covered. However, it is not optimized yet using the minimal length criteria given in (10).

Next we construct the set of FCESs. To do so we examine the set of FEPs. The dashed edges of the CESG in *Fig. 11* represent the FEPs of the function "`Play and Record a CD or Track`" of the RJB. These edges are listed below.

LL, *SL*, *LP*, *PL*, *LM*, *ML*, *SS*, *RP*, *RR*, *RS* **(16)**

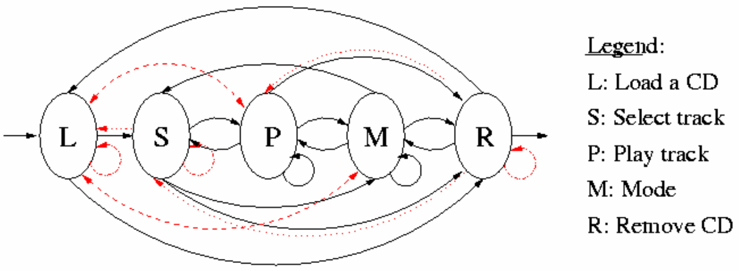


Fig. 11. CESG (Completed ESG) of *Fig. 10* (dashed lines: FEPs)

Based on the algorithm implicit in (4) and the FEPs, we now construct the FCESs systematically in two forms (17):

- FEPs that start at the entry are complete test inputs to trigger undesirable situations, e.g., *LL*, *LP*, *LM* in (16).
- FEPs which do not start at the entry, e.g., *SL*, *PL*, *ML*, *SS*, *RP*, *RR*, *RS* in (14), need prefixes (see (17)).

LL, *LP*, *LM*, *LSL*, *LSPL*, *LSPML*, *LSS*, *LSPMRP*, *LSPMRR*, *LSPMRS* **(17)**

Together with these test inputs, the test process can be carried out as described in (6) and (8).

5.4. Test Generation

As mentioned in Section 5, a set of ESGs was derived manually by studying the user manual of the RJB and through a careful examination of its GUI. The ESGs were input to a test generation tool (see Section 7) to generate CESs and FCESs that constitute the tests for the RJB in this experiment. The tool uses the algorithms sketched earlier and given in Section 4.3 and in Appendix 1. Tests were generated for pair-wise, triple, and quadruple event coverage.

Two student testers applied the generated tests to the RJB semi-automatically using the tools. The testers worked over a period of two weeks, five days a week and, on average, six hours per day, thus spending a total of 60 person-hours. These figures result in approximately 5.5 seconds per test. Faults discovered were noted and analyzed subsequently for their severity.

5.5. Results

Tab. 1 displays the number of tests generated, total count of faults detected, and the cost of detecting a fault measured as faults detected per test case. As shown, a total of 78,611 tests were generated for different n-tuple event coverage using the CES-based and FCES-based test generation.

Also, a total of 68 faults were detected when the RJB was tested against these tests. The number of faults detected increased from 44 through 56 to 68 as n was raised through the values 2, 3 and 4 in n-tuple coverage. While the tests that cover all event pairs reveal 44 errors, coverage of event triples, and then event quadruples, leads to the detection of only 12 new errors, in each case. As indicated in the table, there is a decrease in the number of faults detected per test case 0.0104 to 0.001 for n=2 and 4, respectively. *Fig. 12* shows a plot of the cumulative count of faults detected versus the count of tests generated for n-tuple event coverage.

Tab. 1. Size of the test set and its fault detection effectiveness measured against n-tuple coverage.

Event-tuples covered (n)	No. of test cases	Total count of faults detected	Total count of faults detected per test case (Costs)
2 (pairs)	4,236	44_{total}	0.0104
3 (triples)	10,512	$44_{old}+12_{new}=56_{total}$	0.0053
4 (quadruples)	63,863	$(44+12)_{old}+12_{new}=68_{total}$	0.0010

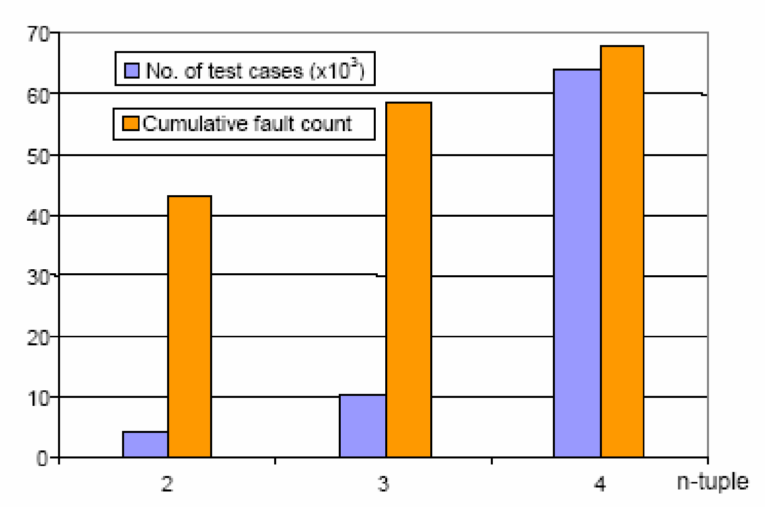


Fig. 12. Number of test cases and the cumulative number of faults detected vs. length of test cases.

Tab. 2 displays the number of tests generated, total count of faults detected, and the cost of detecting a fault measured as faults detected per test case separated by the method of generation, i.e., derived from ESGs and CESGs. The cumulative count of faults discovered from the two sets of tests is shown in *Fig. 13*.

Tab. 2. Size of the generated test sets, fault detection effectiveness, and cost for tests generated from ESGs and CESGs.

Event-tuples covered	No. of test cases derived from ESGs (***Te***)	No. of test cases derived from CESGs ***Tce***)	Total count of faults detected by test cases derived from ESGs (***Fe***)	Total count of faults detected by test cases derived from CESGs (***Fce***)	Cost (***Fe/Te***)	Cost (***Fce/Tce***)
2 (pair)	727	3,509	24_{total}	20_{total}	0.0330	0.0057
3 (triples)	3,672	6,840	$24_{old}+7_{new}$ $=31_{total}$	$20_{old}+5_{new}$ $=25_{total}$	0.0084	0.0037
4 (quad-	27,882	35,981	$(24+7)_{old}+4_{new}$	$(20+5)_{old+}8_{new}$	0.0013	0.0009

ruples)			=35$_{total}$	=33$_{total}$		

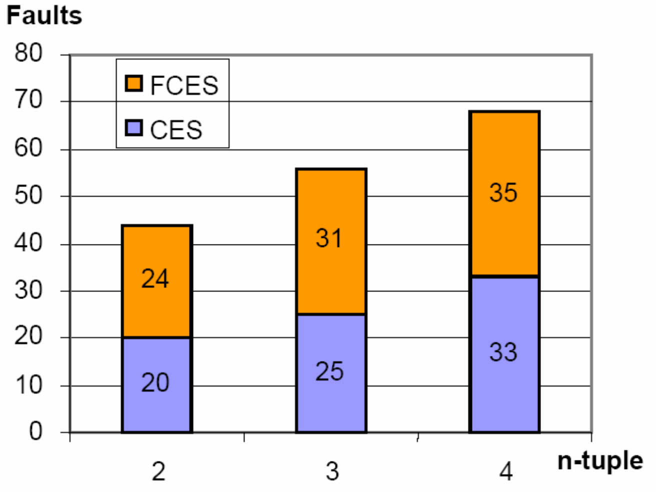


Fig. 13. Fault detection effectiveness of test cases based on CES/CFES versus event-tuple coverage.

5.6. Analysis

We make the following observations from the data in *Tab. 1* and *Tab. 2*.

- Test cases derived for pair-wise coverage, are the most cost-effective when compared with tests that cover triples and quadruples, respectively.
- A rapid decline in test effectiveness is observed with the increasing length of the event sequences used as test cases (cf. *Tab. 1*).
- The test cases derived from ESGs show a higher degree of fault detection effectiveness than those derived from CESGs. This might have been caused, however, by the fact that the SUT has a good exception handling mechanism, even though it is not perfect. (cf. *Tab. 2* and *Fig. 12*).

The test effectiveness measured in terms of the cost per detected fault does not strongly correlate with the event-tuple coverage of the test cases derived from ESGs. The same is true for the tests derived from CESGs. This observation has cost implications for test management as the length and the number of the test cases generated directly effect the cost of testing.

The observation above leads to the recommendation that it is cost-effective to test a system starting with tests derived from the ESGs that cover only the event pairs. If the cumulative number of detected faults grows slowly, then one might terminate the test at this point. Depending on the testing budget, one might then consider generating and executing tests from ESGs that cover event triples and quadruples. The same incremental approach seems appropriate for testing exception handling code using tests generated from CESGs.

5.7. Fault detection

To demonstrate the fault detection capability of the approach, the function `Play track` is analyzed. This function is represented by the ESG in *Fig. 14* as a refinement of node `P` in *Fig. 10* and *Fig. 11*.

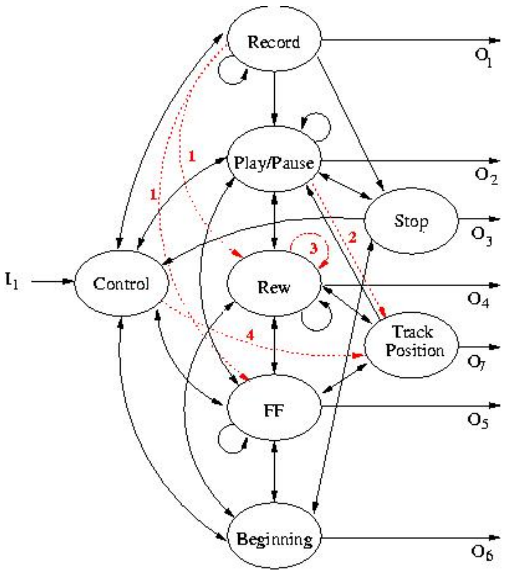


Fig. 14. Completed ESG as a refinement of *Fig. 10*

Some of the detected faults are listed in *Tab. 3*. The fault detection process is simple. As an example, to detect fault 1 in *Tab. 3*, one starts with the `Control` option of the Main Menu of the RJB as in *Fig. 10* and sequentially pushes the button `Rec` and then the button `Rew`, or alternatively, the button `FF`, as shown in *Fig. 14* as alternative edges labelled with No. 1. The other faults in *Tab 3,* labelled accordingly in *Fig. 14* with fault numbers, can be detected similarly. A complete list of the faults detected is included in Appendix B, which also includes the test sequences that reveal these faults. This list includes also the type of the fault, as classified in Section 3.4.

Tab. 3. Detected faults related to the system function `Play track` (node P in *Fig. 10*)

No.	Faults Detected by the ES
1.	While recording, pushing the `forward` button or `rewind` button stops the recording process without due warning.
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.
3.	Menu item `Play/Pause` does not lead to the same effect as the control buttons that are sequentially displayed and pushed via the main window. Therefore, pushing `play` on the control panel while the track is playing stops the playing.
4.	`Track position` could not be set before starting to play the file.

5.8. Defense Mechanism, Risk Ordering

Once the system has been transferred into a faulty state, it cannot accept any further legal or illegal inputs, because an undesirable situation can neither be moved to a desirable one, nor can it be transferred into an even more undesirable one. This level of abstraction ignores fault propagation, whereby faulty events could lead to other faults, possibly, of greater severity. Therefore, prior to further input, the system must recover, i.e., the illegal interaction must be undone by moving the system into a legal state through a backward, or a forward, recovery mechanism (Goodenough, 1975; Randell, 1978).

The construction of the FCESs as described in (4) guarantees that only their last two symbols (as an FEP) are incompatible, in other words, for the determination of the position in which a correction can take place, backtracking by only one symbol is necessary. Having backtracked, possible modes of recovery (i.e., corrections) depend solely on the number of the different symbols which can then transfer the system into a correct state. In this sense, as an example, the faulty event sequence *SL* in (16) is “less risky” in terms of flexibility in fault correction than the sequence *LL*, that is, *SL* □ *LL,* This is because

- *LL* can be transferred to the only two legal event pairs *LS* and *LR* after backtracking to *L*,
- while *SL* can be transferred to three legal event pairs *SP*, *SM* and *SR* after backtracking to *S*.

Thus, for self-correction, any FCES that includes *SL* as a FEP represents a situation which is “less risky” (more desirable) than an FCES of the same length that includes *LL*, for example, in order to automatically navigate the user despite his/her faulty input.

The risk ordering relation can be implemented by incorporating it into a conventional parsing algorithm known in compiler construction (Aho et al., 1977). In uncritical cases a forward recovery might be more convenient, e.g., wherever possible, alerting immediately the user when an FEP is detected, and continue the operation to reach a safe state.

The extended fault model (Section 3.5.4) can be extremely useful for forming fault hypotheses that take the individual risk ratios into account.

5.9. Discussion

At first glance, it seems that only 68 faults are detected upon executing approximately 80,000 test cases (*Tab. 1*) – a huge test effort to detect a relatively small number of faults, especially compared to the previous experience (Belli, 2001) with the same approach. However, the following circumstances appear to explain the situation and to justify the work:

- The algorithms used for determining MSCES and MSFCES (Section 4.3.1 and 4.3.2) have not been considered here as a way of reducing the number of tests. This is because their use would have concealed the number of faults detected by any sequence depending on its length, as well as the number of the individual test cases that would have been covered by a single MSCES or by a single MSFCES.
- Due to intensive and extensive deployment over many years, the product subjected to test is of high quality.
- Given the above, it was encouraging to note that the approach could detect faults at all in this product. This motivates us to further refine and improve the proposed approach.

A key conclusion is that the approach facilitates a simple, but nevertheless a cost-effective, stepwise and straightforward test strategy. This is because it enables the enumeration of test cases (based on the CES and CFES) and, thereby, helps manage the scalability of the test process.

6. Case Study 2: Railway Crossing

In this section, an additional case study sketches the versatility of ESG for modeling and analysis.

6.1. Objectives

The objective of this analytical study is to demonstrate the application of the proposed ESG-based approach for modeling and risk analysis in the area of safety critical systems. For this purpose we considered a simple railway crossing as an example. Though the ESG model generated in this study can be used to generate tests that are use for testing a simulation model of the safety critical system, this was not an objective of the study.

Railway crossings of the kind considered here are found across minor roads outside of towns. They often consist of a pair of gates and two traffic lights: red and green, and also a railway signaling system to control the train movement in the proximity of the crossing, though the latter is ignored here for simplicity. Note that in this model the human is a part of the system environment, e.g., as a driver, a gate controller, etc. The ESG-based approach enables the consideration of both the expected, i.e., correct, and faulty behavior of the human operator. Despite its simplicity, the example is sufficiently expressive for the purpose intended here. Note, however, that the discussion is based on an ordinary familiarity of the application and, therefore, the representation may not be quite accurate from a specialist's point of view.

6.2. System model

An ESG model of such a crossing is shown in *Fig. 15.* The set of input signals (or events) α (see (1)) are partitioned into the subsets α_{sys} and α_{env} with

- $\alpha_{sys} = \{R, G, C, O\}$ as *system signals* and
- $\alpha_{env} = \{T, V\}$ as *environmental events* detected by a system that monitors the crossing.

Here, R denotes the traffic signal turning red, G the traffic signal turning green, C gate closing barring vehicular traffic, as well as other road users, from using the crossing, O for gate opening allowing vehicle traffic through, T train passing the crossing, and V for a vehicle using the crossing. These events bring about hazardous states posing different risks to road and train users. The nature of these hazards varies from state to state of the railway crossing system, some posing greater threats than others. For example, compared to the safest possible state in which all traffic lights are red, the state in which the gate is open carries a great risk since the road users are now free to cross the junction, exposing themselves to danger from a passing train. Likewise, the state in which a train is crossing the junction poses a greater risk than the state in which the gate is closed as the latter includes also situations when there is no train at the crossing. The example, as modeled in *Fig. 15*, assumes that the lights turn green "on demand," that is, when a vehicle reaches the barrier. Once the lights are changed from red to green, they cannot be returned to red until at least one vehicle has passed.

Fig. 15 also indicates the relative risk levels brought about by the occurrence of the different events. In the ESG, the events posing greater threats to the users of the system are placed vertically higher than those posing relatively lower risks. In this respect, it is important to note the significance of placing the events T and V in α_{env} at the top in *Fig. 15*. This is because they denote, in effect, human actions, including potentially erroneous actions. Note also that, as a simplification, the above representation does not include any means to control the movement of trains and the system is assumed to be initialized with a sequence of signals *RC*.

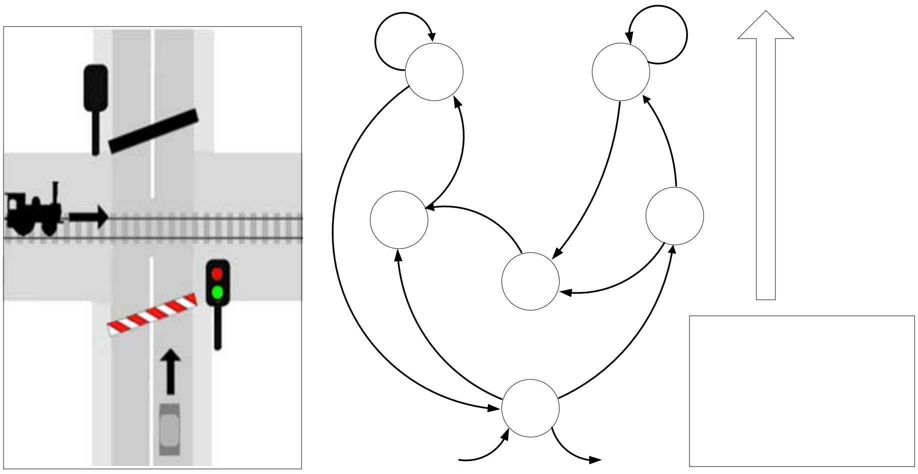


Fig. 15. An ESG model of a railway level crossing

6.3. System Functions and Malfunctions

As is shown by directed arcs in *Fig. 15*, the event pairs in this example are

RG, GV, VV, VR, RC, CT, CO, TT, TO, OG **(18)**

while the complete event sequences (CESs) in any complete cycle of system operation can be represented by the regular expression

$$RE = (RGV^{+})^{*}R+((RGV^{+})^{*}RCT^{*}OGV^{+})^{*}R = ((RGV^{+})^{*}\ (\lambda + RCT^{*}OGV^{+}))^{*}R \qquad \textbf{(19)}$$

where $\boldsymbol{\lambda}$ denotes the empty event sequence. The faulty event pairs for the railway crossing are generated from the complete ESG shown in *Fig. 12*. The FEPs are:

RR, OO, CC, GG, RO, RT, RV, OR, OC, OT, OV, CR, CV, CG, TR, TC, TV, TG, VO, VC, VT, VG, GO, GC, GR, GT **(20)**

The FEPs are shown as dashed lines in the CESG in *Fig. 12* while the EPs are shown as solid lines. In the context of the framework introduced in Section 3, the expression *RE* above constitutes the *system function F*, while those in (20) the system malfunctions ***D.*** We refer to the elements of *D* also as vulnerabilities. Each FEP in (20) represents the leading pair of signals of an emerging faulty behavioral pattern, with the first event being an acceptable one and the second an unacceptable one.

Should the first event of any of the FEPs, e.g., *RV*, matches the last event in any of the ESs, e.g., in $(RGV^{+})^{*}R$, then concatenation of the corresponding ES and the FEP, e.g.,

$$(RGV^{+})^{*}RV \qquad \textbf{(21)}$$

describes, or signifies the occurrence of, a specific form of a faulty behavioral pattern.

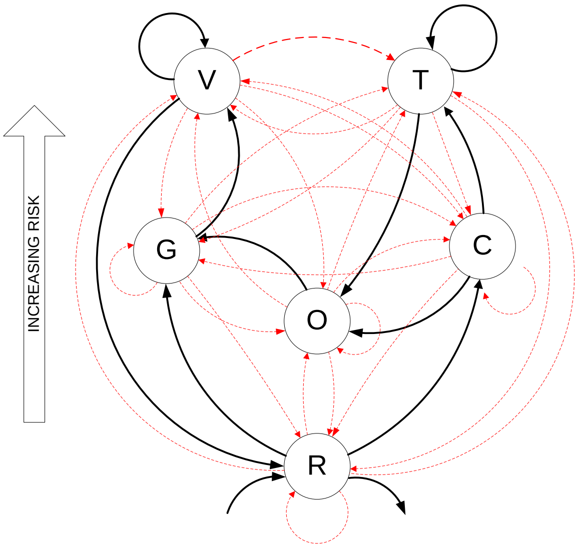


Fig. 16: A CESG model of the railway level crossing with FEPs (dashed lines: FEPs)

Concatenation of the corresponding pairs of ESs and FEPs in the appropriate manner (i.e., by dropping either the last signal of the EP or the first signal of the FEP) results in expressions not belonging to the language described by *R(M)* for system *M*. *Tab. 4* lists the pairs of event sequences and vulnerabilities for the railway crossing together with their interpretations. In spite of its simplicity, the interpretations of the conjunctions of the appropriate pairs (ES, FEP) demonstrate the effectiveness of the approach in revealing the safety-critical situations. Note that for brevity not all FEPs have been considered in *Tab. 4*.

Tab. 4. Level crossing vulnerabilities, the level of the threats posed, and possible defense actions

ES (Column 1)	FEP (Column 2)	Interpretation (Column 3)	Comment (Column 4)	Defense action (Column 5)
$(RGV^{+})^{*}R$	*RO*	Gate opens while lights are set to red (No effective state change is possible except immediately after initialization when the gate was closed).	Ignored	–
	RT	A train arrives prematurely.	Danger	*RC*
	RV	Vehicle traffic passes through red lights.	Danger	†
$(RGV^{+})^{*}RC$	*CR*	Lights to revert to red, though already red.	Ignored	–
	CV	Vehicle traffic is attempting to cross the closed gate and the red lights.	Danger	†
	CG	Lights turn green from red while the gate is closed.	Danger	†
$(RGV^{+})^{*}RCT^{+}$	*TR*	Lights to revert to red while already in red.	Ignored	–
	TC	Gates to close while already closed.	Ignored	–
	TV	Vehicle traffic crosses as trains pass.	Potential	None

			accident	
	TG	Lights turn green as trains pass.	Danger	
$(RGV^{+})^{*}RCT^{*}O$	OR	Lights to revert to red while already in red.	Ignored	–
	OC	Gates to close while already closed.	Ignored	–
	OT	A train arrives after the gate opened.	Danger	RC
	OV	Vehicle traffic crosses as soon as the gate opened but before the lights change to green.	Danger	†
$(RGV^{+})^{*}RCT^{*}OG$, RG	GO	Gates to open though already opened.	Ignored	–
	GC	Gates to close after the lights turn green.	Annoyance	
	GT	A train arrives soon after the lights turn green.	Danger	RC
	GR	Lights turn red before vehicle passes.	Ignored	–
$(RGV^{+})^{*}RCT^{*}OGV^{+}$, RGV^{+}	VO	Gates to open though already opened.	Ignored	–
	VC	Gates to close while vehicle traffic moving.	Danger	VR
	VT	A train arrives amidst vehicular traffic.	Potential accident	RC
	VG	Lights to turn green though already green.	Ignored	–

† - Any defense action is outside the scope of the current model due to lack of features for controlling train movements.

6.4. Defense Mechanism and Risk Graph

To overcome the possible ambiguity in the descriptive nature of *Tab. 3*, a risk graph as in *Fig. 17* may be used. The graph expresses the relative risk levels of states with a greater formality and precision. Each node in the risk graph represents a state that is reached when a given sequence of events occurs. The set of event sequences that bring the system to a given state is indicated as a regular expression in the risk graph.

Using the notation as in (5), a directed edge from a state some s_1 to s_2 in *Fig.13* is equivalent to s_1 □ s_2, signifying that the risks posed by the state s_2 is known to be at the same level as, or exceed, the risks posed by s_1. As a convention in the risk graph, an upward pointing edge signifies that the state lying above poses a greater risk than the one lying below. Arcs drawn in solid lines, as well as the states denoted by underlined regular expressions, refer to the normal functional behavior, while those with dashed lines and other (non-underlined) FES (regular) expressions refer to vulnerability states. To reduce clutter, the diagram does not show the reflexivity of the permissions in the relation □ (i.e., loops at nodes) and shows the vulnerability states used for demonstrative purposes only, i.e., it is not complete concerning FEPs. In this particular case, the last event of most regular expressions, describing a vulnerability state pointed at by a dashed arc, denotes a human action, such as operating a train. More strikingly, each and every state, lying highest in the diagram, is described by a regular expression ending with one of the two events *T* and *V*, each related to a human action of operating a train or a vehicle, respectively. Therefore, in addition to the risks associated with the functional behavior, the risk graph allows a way to represent explicitly the risks associated with potential human errors.

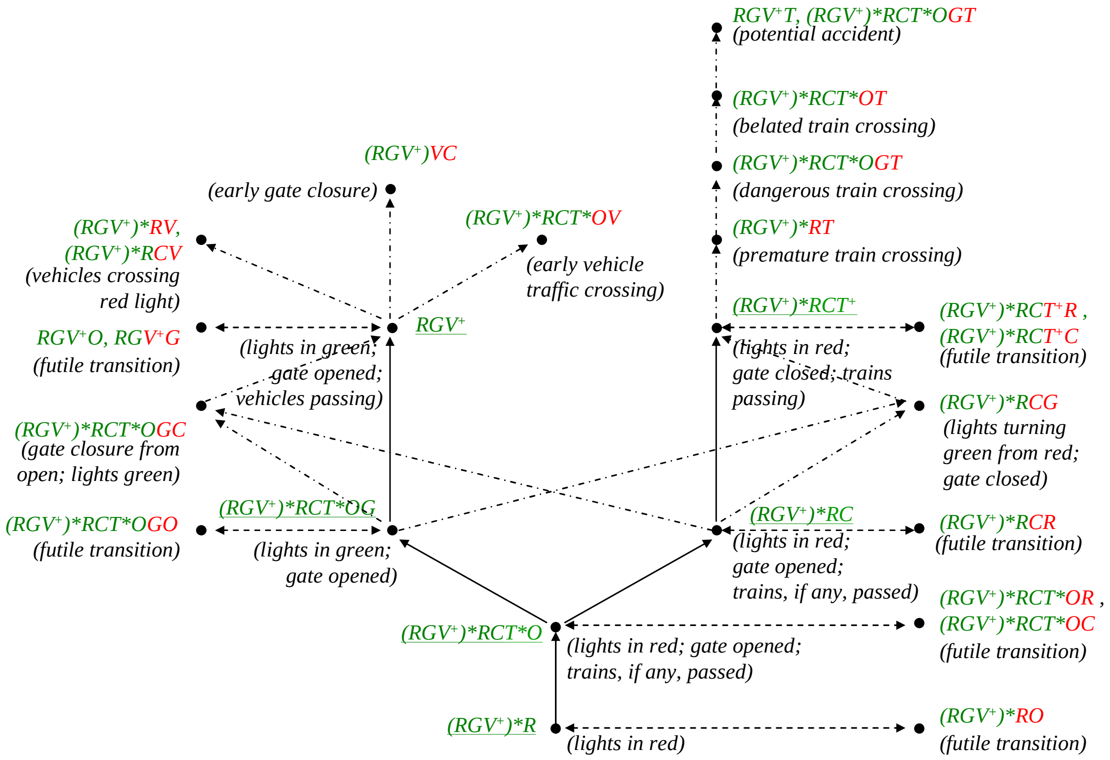


Fig. 17. Risk graph of the railway crossing, covering both the system and vulnerability states

Having identified potential vulnerabilities, it is possible to provide measures that counteract them. This is the intention of the *defense matrix* and exception handlers. In this connection, an attempt is in *Fig. 17* to propose the defense actions that may be taken. Due to the limited scope of the model, these actions only partially address the potential vulnerabilities. This is because all defense actions at the disposal of the current model are limited to closing the gate or turning the traffic lights to red, thus affecting only the vehicle traffic.

A richer model with features for modeling signaling mechanisms would allow the means to address other vulnerabilities, namely, those that can be avoided or mitigated by controlling the train movements. Should *Tab. 3* be complete in these respects, the event sequences listed under column 5 would be equivalent to the set of the exception handlers X, while the columns 1, 2 and 5 would amount to a definition of the required defense matrix implicitly, provided that the data in these columns satisfies the condition in (7). Note that the concatenation of expressions in columns 1, 2 and 5 in the appropriate manner (i.e., by dropping common events as appropriate) gives the state aimed at by the defense matrix as a result of invoking the corresponding exception handler (Leveson, 1986; IEC 61025; IEC 61508).

6.5. Testing Issues

The test process can now be worked out analogous to that described in (6) and (8) in Section 3.2. Thus, the CES and FCES are systematically constructed and combined to cover the edges of the CESG as demonstrated in the introductory example in Section 3, i.e., CES and FCES are input to trigger desirable, and undesirable, situations, respectively.

In the case of an application such as a railway level crossing, testing of the application in requires a simulation model. Such a model could be in software or a mix of hardware and software. As an example, the test input (21) represents the event that the vehicle traffic passes through the red lights, which cannot be realized as a real-life experiment. Furthermore, in order to generate a complete test case, a meaningfully reactive controlling system is needed, which is outside the scope of the current model, given the representation in *Fig. 15*.

Nevertheless, even this simple model is useful in that it makes such dangerous situations explicit (visible) and highlights the reactions required of the controlling system in response to such inputs. Thus, it is evident that one can use the CESG in *Fig. 16* to simulate all potential test scenarios.

To avoid unnecessary details, the results of the analysis are summarized below covering all edges of the CESG given in *Fig. 16*. It appears that following sets of event sequences, ESs, are of particular interest when dealing with system vulnerabilities:

$$(RGV^{+})^{*}R,\quad (RGV^{+})^{*}RC,\quad (RGV^{+})^{*}RCT^{+},\quad (RGV^{+})^{*}RCT^{*}O,\quad (RGV^{+})^{*}RCT^{*}OG,\quad (RGV^{+})^{*}RCT^{*}OGV^{+},\ (RGV)^{*}RCT^{*}OGV^{+}R \qquad \textbf{(22)}$$

These ESs are possible prefixes, i.e., starters, that can be constructed by analyzing the expression (19). The test inputs can be now constructed as described in the previous section. For example, it is possible to generate *RGVRV* as an instance of the sub-string $(RGV^{+})^{*}RV$. A correct implementation of the railway crossing controller should respond to this test by the defense action *RC* (see *Tab. 3*). Thus, the particular test input *RGVRV* is designed to test the system response to a simulation of a human error, that is, driving a train through the level crossing prior to closing the gate, despite the signals having turned red. Other kinds of human errors, particularly those related to poor user interface design (Redmill and Rajan, 1997; Shneiderman, 1998), may be addressed in a similar manner.

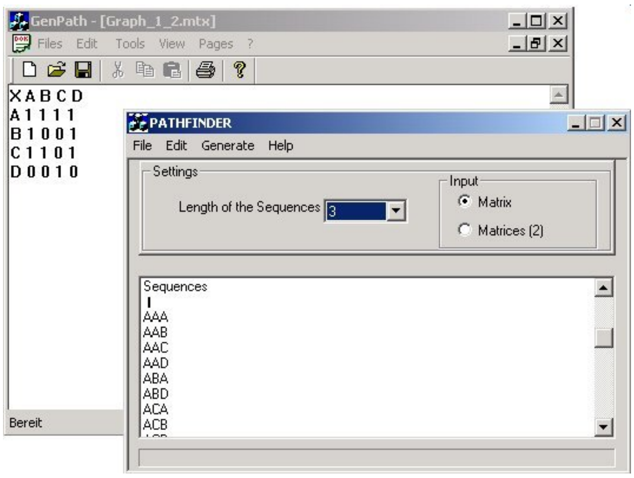
7. Tool Support

The generation of test cases from ESG and CESG and the determination of the MSCESs and MSFCESs can be arduous and time consuming if done manually. Here we describe a toolkit that automates the test generation and viewing process.

7.1. Test Case Generation

GenPath is a tool we developed for the generation of test cases, i.e., ESs and FESs to obtain a given n-tuple event coverage. The tool takes the ESG as input in the form of an adjacency matrix, or as a regular expression. Several ESGs, that form a hierarchy, can be input together. In this case an ESG at a lower level in the hierarchy will be a refinement of a node in a higher level ESG. *Fig. 18* represents the topmost screen of the GenPath tool including the adjacency matrix of an example ESG.

Shown on the right hand side are the sequences of length 3 for the same given ESG, generated subsequently by another tool *PATHFINDER*.



```
X A B C D
A 1 1 1 1
B 1 0 0 1
C 1 1 0 1
D 0 0 1 0
```

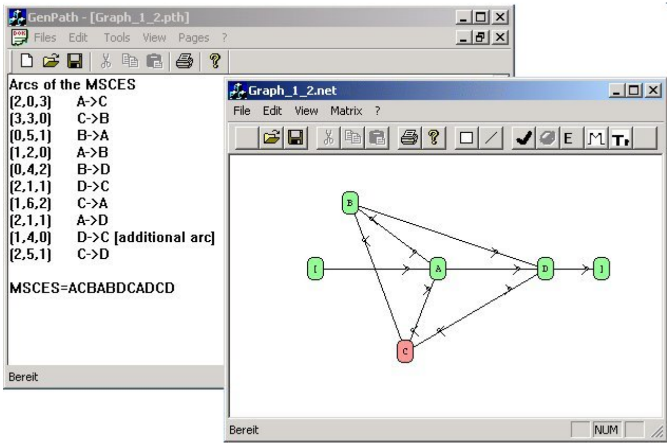
Length of the Sequences: 3

Input: ● Matrix ○ Matrices (2)

```
Sequences
I
AAA
AAB
AAC
AAD
ABA
ABD
ACA
ACB
```

Fig. 18. Tool GenPath; mode to generate test cases

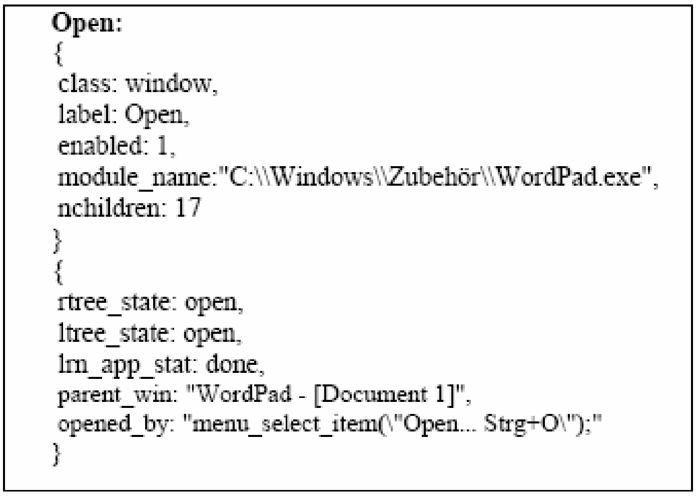
GenPath also generates MSCESs given a n-tuple coverage requirement. In addition, it displays the ESG under consideration and marks its EPs (*Fig. 19*).



```
Arcs of the MSCES
[2,0,3]    A->C
[3,3,0]    C->B
[0,5,1]    B->A
[1,2,0]    A->B
[0,4,2]    B->D
[2,1,1]    D->C
[1,6,2]    C->A
[2,1,1]    A->D
[1,4,0]    D->C [additional arc]
[2,5,1]    C->D

MSCES=ACBABDCADCD
```

Fig. 19. GenPath to generate MSCES

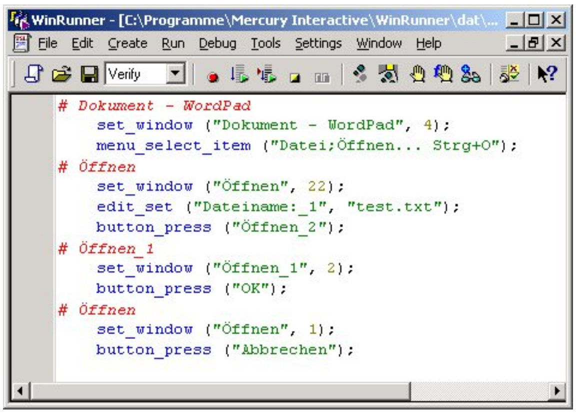


```
Open:
{
class: window,
label: Open,
enabled: 1,
module_name:"C:\\Windows\\Zubehör\\WordPad.exe",
nchildren: 17
}
{
rtree_state: open,
ltree_state: open,
lrn_app_stat: done,
parent_win: "WordPad - [Document 1]",
opened_by: "menu_select_item(\"Open... Strg+O\");"
}
```

Fig. 20. Excerpt out of the WinRunner GUI-file with information on the GUI component "Open"

7.2. Generation of Test Scripts

We have seen how regular expressions are used as a compact representation of a set of event sequences. A regular expression is considered here as representation of *test scripts*. Consideration of specific information of the components of the SUT is necessary in order to construct test scripts that are used in generating test cases. This information can be obtained manually by writing appropriate test scripts, but this process can, however, be tedious and error-prone since the user has to know all commands of the test script language. The deployment of a tool with a Capture-Playback facility suits the purpose better, and can be commercially obtained, e.g., WinRunner of Mercury Interactive (*Fig. 21*). Such tools can identify all available windows of a GUI.


WinRunner - [C:\Programme\Mercury Interactive\WinRunner\dat\...
File Edit Create Run Debug Tools Settings Window Help
Verify

```
# Dokument - WordPad
    set_window ("Dokument - WordPad", 4);
    menu_select_item ("Datei;Öffnen... Strg+O");
# Öffnen
    set_window ("Öffnen", 22);
    edit_set ("Dateiname:_1", "test.txt");
    button_press ("Öffnen_2");
# Öffnen_1
    set_window ("Öffnen_1", 2);
    button_press ("OK");
# Öffnen
    set_window ("Öffnen", 1);
    button_press ("Abbrechen");
```

Fig. 21. Generated Test Script

The captured properties of each component are stored in a GUI-file of WinRunner which can be used to generate appropriate test inputs for the corresponding test cases. *Fig. 21* represents a part of WordPad that the test environment has traced.

8. Discussion

We have introduced an integrated approach to the modeling, analysis, and test generation for embedded, real-time, and interactive systems. Modeling is carried out using event sequence graphs (ESGs). These graphs and their complements assist with the verification of the expected system behavior in the presence of expected inputs as well as the analysis of risks associated with system behavior in the event of failure of its exceptional or safety handling mechanisms in unexpected situations.

Two studies presented indicate how ESGs can be used to model and analyze the behavior of a system. We have also shown how the ESG-based model is used for test generation. The effectiveness of the tests so generated is reported. An ESG-based model allows incorporation of both the desirable and undesirable features of the system. The degree of undesirability is represented in the form of a risk ordering relation – an expression of relative levels of risks posed by hazardous states. This allows targeting the design of tests at specific system attributes.

The fault model in this work has been intentionally kept simple: states, inputs and outputs of FSA have been merged into the vertices of an ESG and its complement $\overline{ESG}$, which are uniformly interpreted as events; with no annotation of edges of either. The test process based on this model generates test cases as sequences of edges of the ESG and $\overline{ESG}$ to test whether or not the system behaves as desired and is robust in the face of interactions with faulty inputs.

As ESG (and thus $\overline{ESG}$) is constructed to reflect the user expectations, the acting as an oracle of a high level of trustworthiness, de facto resolving the oracle problem. Furthermore, criteria are developed to determine the completeness of the test process, enabling a decision on when to stop testing. These criteria, based on the coverage of edges of the underlying ESG and $\overline{ESG}$, are used to construct a minimal set of test cases.

8.1. Advantages and Disadvantages of Modeling with ESG

The concept of ESG as a simplification of FSA enables the exploitation of the features of the type-3 languages, including decidability results on the recognition problem (necessary for effectively complementing the ESG), well-known algorithms used in automata testing, and compiler construction, e.g., for handling faulty programs (see Section 5.8). The trade-off between this simplification and elegance achieved through ESGs is that it neglects the states of the SUT and the hierarchical levels of the user interactions.

Generation of test cases that rely on information about the internal behavior of the system might be difficult to achieve with ESGs. An example is a test designed to check that a save operation is not executed if the loaded file is write-protected. Another is a test designed to check that a button has not been deactivated inadvertently by a previous operation offered by a menu with many entries for alternative user inputs. Presentation of such situations with ESGs is generally possible, but could become tedious. In the latter example, for instance, the likely combinations of different values of corresponding flags, which could have been set or reset in different menus, could be numerous. In all these cases, Boolean algebra-based techniques, such as decision tables and Karnaugh-Veitch diagrams, might be more convenient alternatives for constructing test cases (Binder, 2000). As in any problem solving activity, there may not be a “silver bullet” type single test that can cope with every kind of fault.

8.2. Recommendations for Practice

The ESG-based approach has been applied to the testing of the GUIs of different industrial applications; e.g., the GUIs of a mobile telephone device, a ticketing machine, etc. (Belli, 2001). In addition, the approach has also been used to validate requirements definitions and to verify and design specifications, both mainly represented by ESGs. While some of the results of the analysis of the detected faults were in compliance with the expectations, other results were surprising, and are summarized below.

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” (Goodenough, 1975) 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%. Poor handling of exceptions has also been reported by Westley and Necula (Wesley, 2004).

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.

9. Future Work

If a more sophisticated fault classification model, e.g., Orthogonal Defect Classification (Chillarege, 1996), is required, the fault model must be extended accordingly, differing across states, inputs and outputs. Following the guidelines in Section 3.5, the model extension aims at distinguishing between different kinds of faults and levels of their severity, leading to a general, effective strategy for fault handling, e.g., to determine the test set and costs for a given safety level.

A first step in this direction has been reported (Gutzeit, 2003) by applying the approach introduced in this paper to Statecharts (Harel and Naamad, 1996). Further work is planned to consider UML –

an approach already exploited for generating test cases (Kim et al., 1999; Briand and Labiche, 2002; Offutt, 2003). However, further research is needed to extending the notions and algorithms introduced and summarized in this paper, particularly in relation to state explosion caused by additional nodes while completing the ESG and to account for concurrency in system behavior (Schneider, 1990; Raju and Shaw, 1994).

Experience with the ESG based approach suggests that the number of selected test cases can be reduced by considering structural features of the SUT, e.g., identifying windows that cannot invoke any child windows, or that cannot simultaneously exist with windows of the same hierarchy level, etc. Such *terminal* windows need not be considered combinatorial while generating test cases. This aspect is likely to help in the elimination of unnecessary and/or infeasible test cases and thus in a significant cost reduction. Consideration of further modeling notions, e.g., based on Kripke structures **(**Peled, 2001**)**, may offer further research avenues.

Finally, additional vulnerability attributes are to be considered, particularly in applications that can be modeled in a state-based formulation. These include, for example, security (Eckmann et al., 2002).

References

A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Principles of Compiler Design.* Addison-Wesley, 1977.

A. V. Aho, A. T. Dahbura, D. Lee, and M.Ü. Uyar, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours," *IEEE Trans. Commun*., vol. 39, no. 11, pp. 1604-1615, 1991.

V.R. Basili, R.W. Selby, and D.H. Hutchens, "Experimentation in Software Engineering," *IEEE Trans. On Softw. Eng.*, vol. 12, no. 7, pp. 733-743, 1986.

F. Belli and J. Dreyer, "Program Segmentation for Controlling Test Coverage," *Proc. 8th Int'l Symp. Softw. Reliability Eng. (ISSRE '97)*, pp. 72-83, 1997.

F. Belli and K.E. Grosspietsch, "Specification of Fault-Tolerant System Issues by Predicate/Transition Nets and Regular Expressions – Approach and Case Study*," IEEE Trans. On Softw. Eng.,* vol. 17, no. 6, pp. 513-526, 1991.

F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces," *Proc. 12th Int'l Symp. Softw. Reliability Eng. (ISSRE '01),* pp. 34-43, 2001.

R.V. Binder, *Testing Object-Oriented Systems*. Addison-Wesley, 2000.

G. V. Bochmann, A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," *Softw. Eng. Notes*, ACM SIGSOFT, pp. 109-124, 1994.

L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *Software and System Modeling,* vol. 1, no.1, pp.10-42, 2002.

R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and Man-Yuen Wong, "Orthogonal Defect Classification – Concept for In-Process Measurements," *IEEE Trans. On Softw. Eng.,* vol.18, no. 11, pp. 943-956, 1992.

R.S. Chhikara and R.P. Heydorn, "Event Sequence Diagrams for Dynamic Probabilistic Risk Analysis," *Annual Report of the Institute for Space Systems Operations*, The University of Houston, Clearlake, 1999-2000.

T.S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. On Softw. Eng.,* vol. 4, no. 3, pp. 178-187, 1978.

E.W. Dijkstra, "A Note on Two Problems in Connection With Graphs," *J. of Numerische Mathematik*, pp. 269-271, 1959.

M.E. Delamaro, J.C. Maldonado, and A. Mathur, "Interface Mutation: An Approach for Integration Testing," *IEEE Trans. on Softw. Eng.,* vol. 27, no. 3, pp. 228-247, 2001.

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

R.A. DeMillo and A.J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Trans. On Softw. Eng.,* vol. 17, no. 9, pp. 900-910, 1991.

S.T. Eckmann, G. Vigna, and R. Kemmerer, "STATL: An Attack Language for State-Based Intrusion Detection," *Journal of Computer Security*, vol. 10, no. 1-2, pp. 71-103, 2002.

J. Edmonds and E.L. Johnson, "Matching, Euler Tours, and the Chinese Postman", *Math. Programming*, vol. 5, pp. 88-124, 1973.

B. Eggers, F. Belli, "A Theory on Analysis and Construction of Fault-Tolerant Systems" (in German), *Informatik-Fachberichte*, Springer-Verlag, Berlin, vol. 84, pp. 139-149, 1984.

S.C.P.F. Fabbri, J.C. Maldonado, P.C. Masiero, and M.E. Delamaro, "Mutation Analysis Testing for Finite State Machines," *Proc. 5th Int'l Symp. Softw. Reliability Eng. (ISSRE '94)*, pp. 220-229, 1994.

X. Fetzer and Z. Xiao, "Increasing the Robustness of C-libraries Using Robustness Wrappers," *Proc. of the Int'l Conf. on Dependable Systems & Networks*, 2002.

S. Fujiwara, G.V. Bochmann, F. Khendek, and M. Amalou, "Test Selection Based on Finite State Models," *IEEE Trans. on Softw. Eng.,* vol. 17, no. 6, pp. 591-603, 1991.

A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specification," *Proc. 7th ESEC/FSE '99*, ACM SIGSOFT, pp. 146-162, 1999.

A. Gill, *Introduction to the Theory of Finite-State Machines.* McGraw-Hill, 1962.

J.B. Goodenough, "Exception Handling – Issues and a Proposed Notation", *Comm. ACM,* vol. 18, no. 12, pp. 683-696, 1975.

T. Gutzeit, "Testcase Generation from Statecharts to Validate Graphical User Interfaces" (in German), *TR 2003/6* (Master Thesis), Univ. Paderborn, Angewandte Datentechnik, 2003.

S. Gossens, F. Belli, S. Beydeda, and M. Dal Cin, "View Graphs for Analysis and Testing of Programs at Different Abstraction Levels," *Proc. of High-Assurance Systems Eng. Symp. (HASE 2005)*, IEEE Comp. Society Press, pp. 201-208, 2005.

D. Hamlet, "Foundation of Software Testing: Dependability Theory," *Proc. of the 2nd ACM SIGSOFT Symp. on Foundations of Softw. Eng.*, pp. 128-139, 1994.

D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts", *ACM Trans. Softw. Eng. Meth. (TOSEM),* vol. 5, no. 4, pp. 293-333, 1996.

M.N. Huhns and V.T. Holderfield, "Robust software," *IEEE Internet Computing*, vol. 6, no. 2, pp. 80-82, 2002.

IEC 60300-3-1. Dependability Management – Part 3-1: Application Guide – Analysis Techniques for Dependability – Guide on Methodology.

IEC 61025. Fault Tree Analysis.

IEC 61508. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, especially Part 3: Software Requirements.

IEEE Std. 610.12 1990, IEEE Standard Glossary of Software Engineering Terminology.

ISO/IEC 9126,Software Product Evaluation – Quality Characteristics and Guidelines for Their Use.

K. Jensen and N. Wirth, *Pascal, User Manual and Report.* Springer-Verlag, New York, 1974.

Y.I. Ianov, "Logic Schemes of Algorithms," *Problems of Cybernetics*, Pergamon Press, New York, vol. 1, pp. 82-140, 1960.

Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha, "Test Cases Generation from UML State Diagrams," *IEE Proc. Softw.*, vol. 146, no. 4, pp. 187-192, 1999.

B. Korel, "Automated Test Data Generation for Programs with Procedures," *Int'l Symp. on Softw. Testing and Analysis (ISSTA '96)*, pp. 209-215, 1996.

M.-K. Kwan, "Graphic Programming Using Odd or Even Points," *Chinese Math*., vol. 1, pp. 273-277, 1962.

N.P. Kropp, P.J. Koopman, and D.P. Siewiorek, "Automated Robustness Testing of Off-The-Shelf Software Components," *28th Annual Inter'l Symp. on Fault-Tolerant Computing*, Digest of Papers. pp. 230 – 239, 1998.

N.G. Leveson, "Software Safety: Why, what, and how," *ACM Comp. Surveys*, vol. 18, no. 2, pp. 125-163, 1986.

L. Libeaut, J.J. Loiseau, "Admissible initial conditions and control of timed event graphs," *Proc. 34th IEEE Conf. on Decision and Control*, vol. 2, pp. 2011 - 2016, 1995.

M. Lyu, *Handbook of Software Reliability Engineering*. McGraw-Hill & IEEE CS Press, Los Alamitos, Calif, 1996.

M. Marré and A. Bertolino, "Using Spanning Sets for Coverage Testing," *IEEE Trans. On Softw. Eng.,* vol. 29, no. 11, pp. 974-984, 2003.

A.M. Memon, M.E. Pollack, and M.L. Soffa, "Automated Test Oracles for GUIs," *SIGSOFT 2000*, pp. 30-39, 2000.

A.M. Memon, M.E. Pollack, and M.L. Soffa, "Hierarchical GUI test case generation using automated planning," *IEEE Trans. On Softw. Eng.,* vol. 27, no. 2, pp. 144 - 155, 2001.

J. Myhill, "Finite Automata and the Representation of Events,", *TR 57-624*, Wright Air Devel. Command, pp. 112-137, 1957.

S. Naito and M. Tsunoyama, "Fault Detection for Sequential Machines by Transition Tours," *Proc. of IEEE Fault Tolerant Computing Conference (FTCS)*, pp. 238-243, 1981.

N. Nissanke and H. Dammag, "Design for Safety in Safecharts With Risk Ordering of States," *Safety Science,* vol. 40, no. 9, pp. 753-763, 2002.

J. Offutt, L. Shaoying, A. Abdurazik, and P. Ammann, "Generating Test Data From State-Based Specifications," *The Journal of Software Testing, Verification and Reliability*, vol. 13, no. 1, pp. 25-53, March, 2003.

D.L. Parnas, "On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System," *Proc. 24th ACM Nat'l Conf.*, pp. 379-385, 1969.

D.A. Peled, "Software Reliability Methods," *Texts in Computer Science*, Springer-Verlag, New York, 2001.

R.E. Prather, "Regular Expressions for Program Computations," *The American Mathematical Monthly*, vol. 104, no. 2, pp. 120-130, 1997.

S.C.V. Raju and A. Shaw, A., "A Prototyping Environment for Specifying, Executing and Checking Communicating Real-Time State Machines," *Software – Practice and Experience*, vol. 24, no. 2, pp. 175-195, 1994.

B. Randell, "Reliability Issues in Computing System Design," *ACM Comp. Surveys,* vol. 10, no. 2, pp. 123-165, 1978.

RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification

F. Redmill and J. Rajan, *Human Factors in Safety-Critical Systems*. Butterworth-Heniemann, 1997.

A. Salomaa, *Theory of Automata.* Pergamon Press, Oxford, 1969.

B. Sarikaya, "Conformance Testing: Architectures and Test Sequences," *Computer Networks and ISDN Systems*, vol. 17, no. 2, North-Holland, pp. 111-126, 1989.

F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", *ACM Computing Surveys,* vol. 22, no. 4, pp. 299-319, 1990.

L.W. Schruben, "Building reusable simulators using hierarchical event graphs," *Winter Simulation Conference Proceedings,* pp. 472 -475, 1995.

A.C. Shaw, "Software Specification Languages Based on Regular Expressions," *Software Development Tools*, ed.

W.E. Riddle, R.E. Fairley, Springer-Verlag, Berlin, pp. 148-176 , 1980.

R.K. Shehady and D.P. Siewiorek, “A Method to Automate User Interface Testing Using Finite State Machines,” *Proc. 27th Annual Int'l Symp. Fault-Tolerant Computing (FTCS '97)*, pp. 80-88, 1997.

B. Shneiderman, *Designing the User Interface*. Addison Wesley Longman, 1998.

N. Storey, *Safety-Critical Computer Systems.* Addison-Wesley, 1996.

K. Tai and Y. Lei, “A Test Generation Strategy for Pairwise Testing,” *IEEE Trans. On Softw. Eng.*, vol. 28, no. 1, pp. 109-111, 2002.

H. Thimbleby, “The directed Chinese Postman Problem,” *Softw., Pract. Exper.*, vol. 33, no. 11, pp. 1081-1096, 2003.

W. Westley and G. Necula, “Finding and Preventing Run-Time Error Handling Mistakes,” *Proc. 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pp 419-431, Vancouver, British Columbia, Canada, October 2004.

D.B. West, *Introduction to Graph Theory.* Prentice Hall, 1996.

L. White and H. Almezen, “Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences,” *Proc. 11th Int’l Symp. Softw. Reliability Eng. (ISSRE ‘00)*, pp. 110-121, 2000.

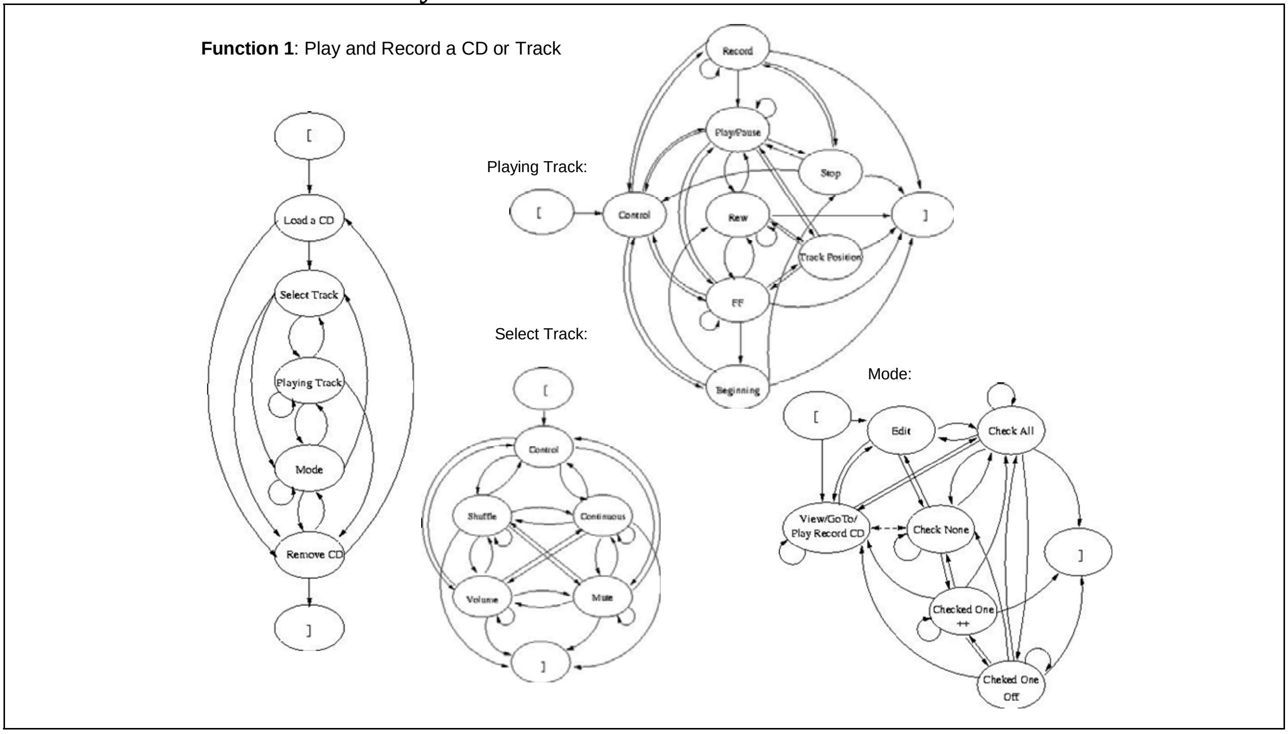
T. W. Williams and K.P. Parker, “Design for Testability - A Survey,” *IEEE Trans. Comp.,* vol. 31, no. 1, pp. 2-15, 1982.

L. Williams, “Formal Methods in the Development of Safety Critical Software System,” *Technical Report No. SERM-014-91*, Software Engineering Research, Boulder, CO, November, 1991.

C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering – An Introduction,* Kluwer Academic Publisher.

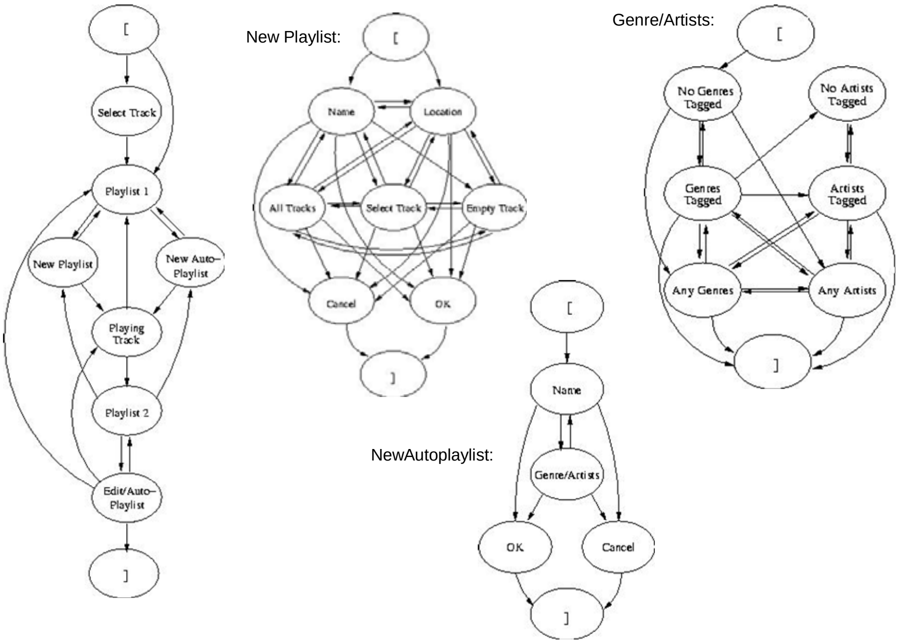
Appendix

A. ESGs of the Case Study 1



Function 1: Play and Record a CD or Track

Function 2: Create and Play a Playlist



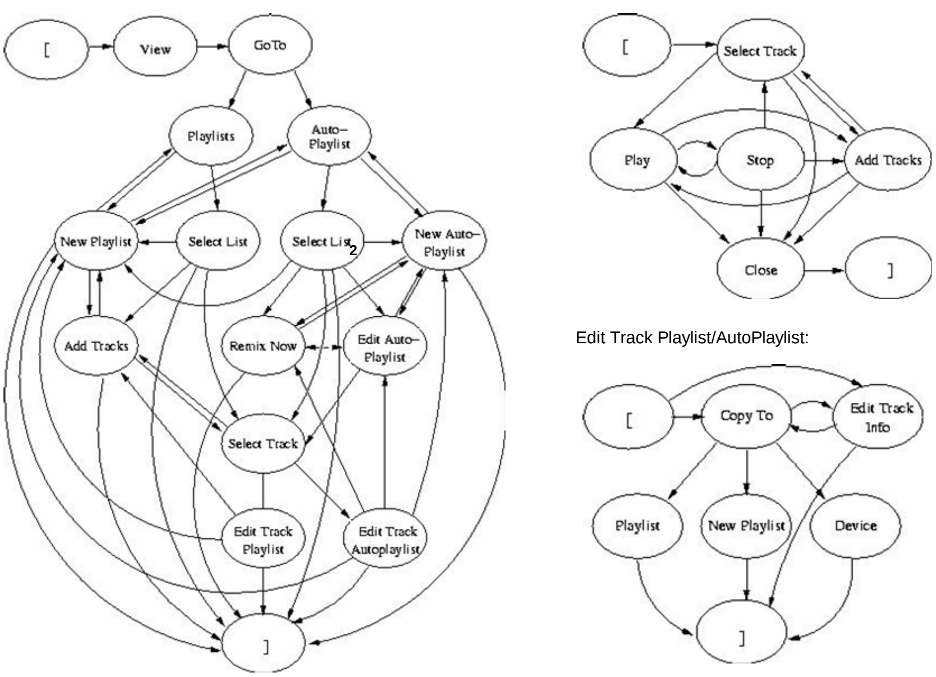
New Playlist:

Genre/Artists:

NewAutoplaylist:

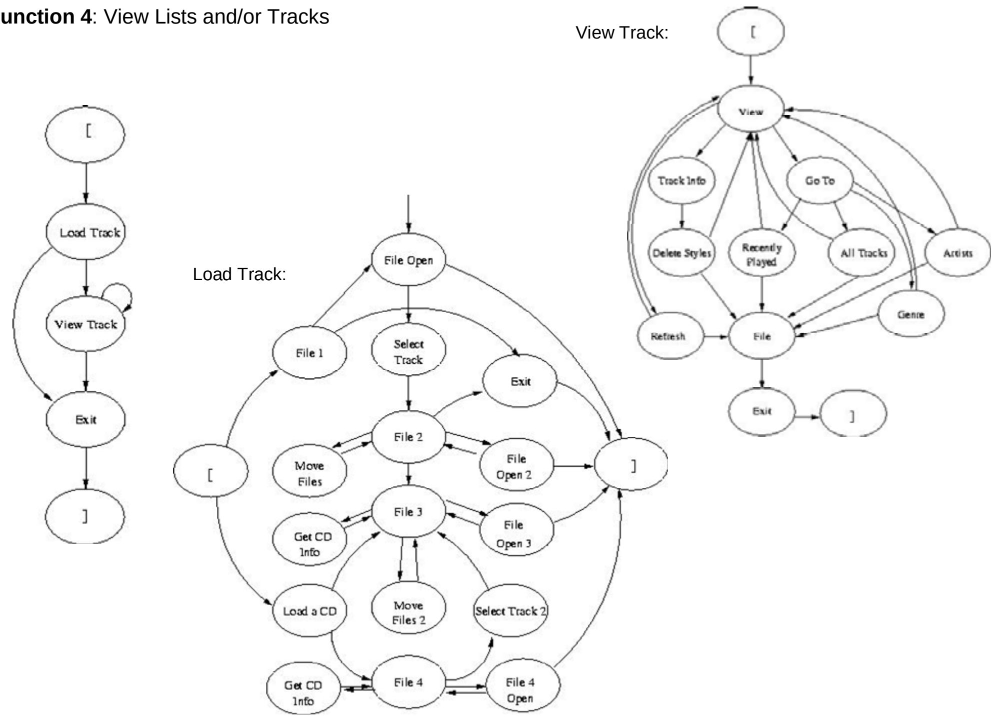
Function 3: Edit Playlists and/or AutoPlaylists

Add Tracks:



Edit Track Playlist/AutoPlaylist:

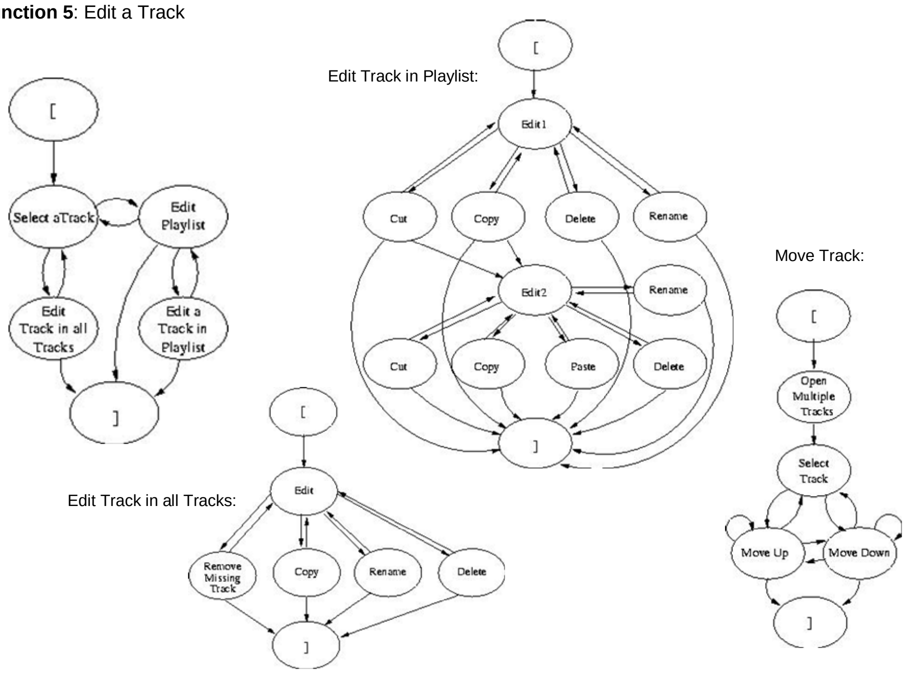
Function 4: View Lists and/or Tracks



View Track:

Load Track:

Function 5: Edit a Track

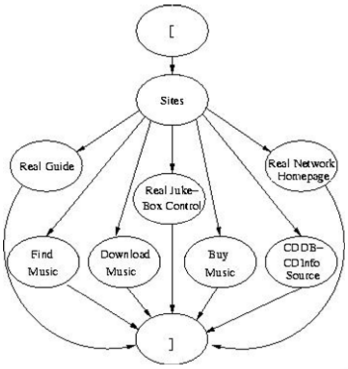


Edit Track in Playlist:

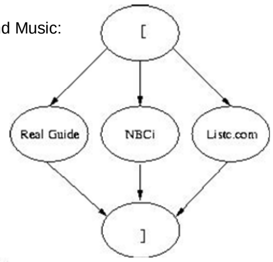
Edit Track in all Tracks:

Move Track:

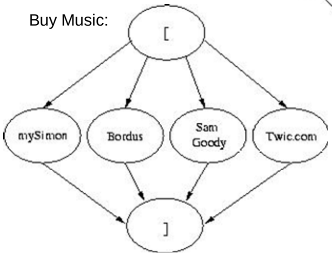
Function 6: Visit the Sites



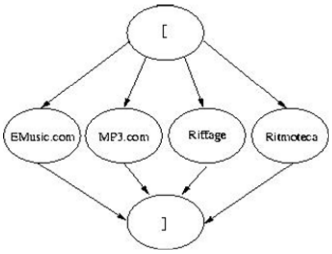
Find Music:



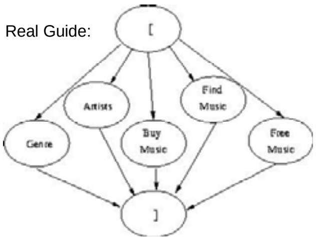
Buy Music:



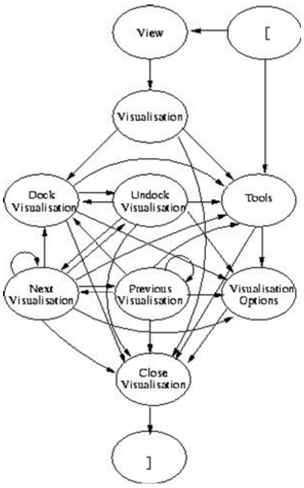
Download Music:



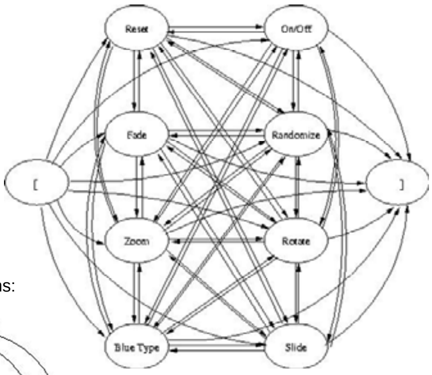
Real Guide:



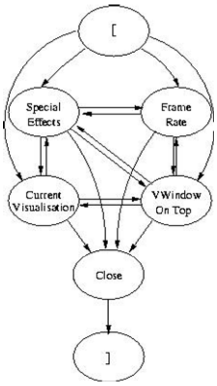
Function 7: Visualization



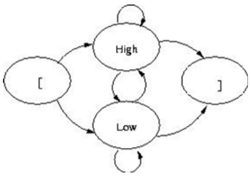
Special Effects:



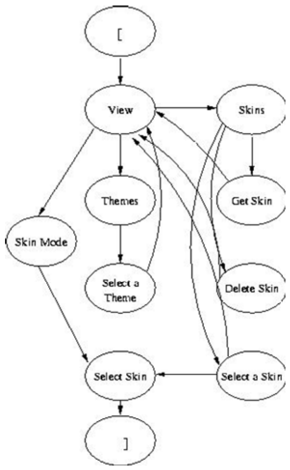
Visualization Options:



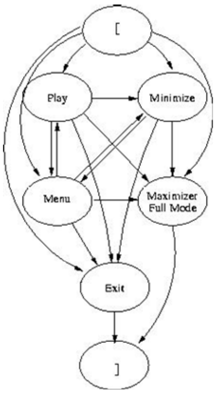
Frame Rate:



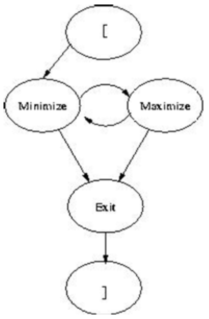
Function 8: Skins



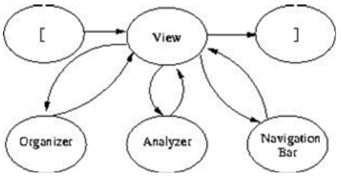
Select Skin:



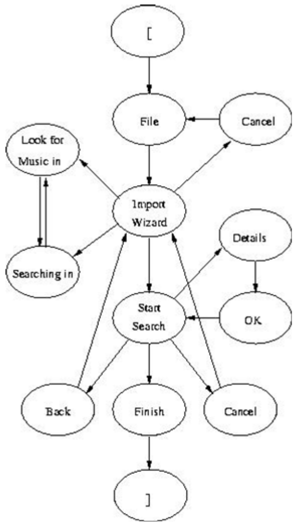
Function 9: Screen Sizes



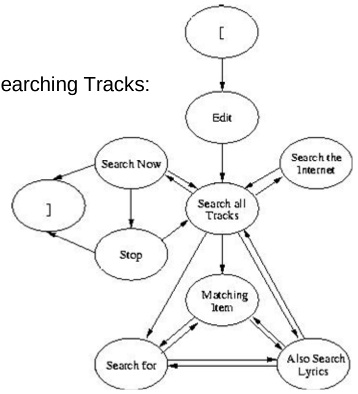
Function 10: Different Views of Windows



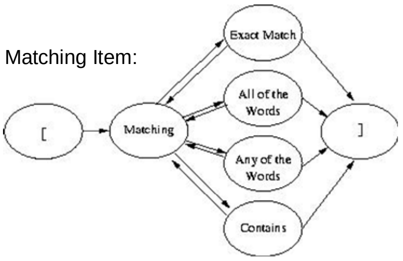
Function 11: Find Music



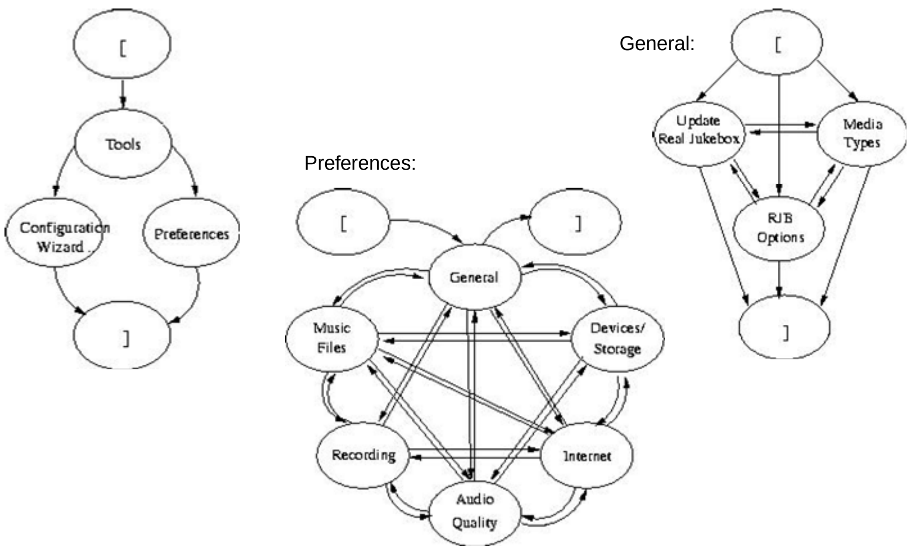
Searching Tracks:



Matching Item:



Function 12: Configure RJB



B. List of Faults Revealed

Function	n-tuple to be covered	Faults detected by CES	Fault Type	Faults detected by FCES
1	2	Shuffle can be enabled when only one track has been selected.	functional	
		Recording an existing track removes (overwrites) the old track from list before recording is completed.	sequencing	
			functional	While recording, it is also possible to forward and/or rewind, causing the recording process to stop.
			functional	Rewind can be activated during playing a CD or a track in shuffle mode.
		Menu item Play/Pause is not the same as the control buttons	functional	

		displayed on the main window. Therefore, pushing start button on the control panel, while the track is playing, stops it.		
		The interaction Play>Pause>Controls>Jump_To>Beginning continues playing the same track while Pause is still displayed.	sequencing	
			functional	Setting the track position, when it is paused, continues playing the file.
		If one track is selected but the arrow shows to another track, hitting play starts playing the selected track.	functional	
		Check one track on/off is not as a menu item available.	functional	
		Track position could not be set before playing the file.	functional	
		Open a file starts playing it.	functional	
			sequencing	Mute button of RJB ignores the situation where the loudspeaker has been reset.
		CD has been removed; RJB ignored this and lists the track names.	sequencing	
		Autoplay cannot start a CD that is already set.	functional	
		Display does not adjust upon inserting a CD, i.e., its content will not be displayed.	functional	
		If another track is played in background, following error message occurs: „ An unknown Error occurred. For more information...”	sequencing	
		Pause is ignored if rewind/fast forward is activated (REW/FF/Track position).	functional	
		Even if pause is activated, beginning starts the track.	functional	

			functional	Track position is disabled when stop is activated.
			functional	Even if Checknone is enabled, Play/Pause/Stop/Rew/FF/Record/Beginning can be activated.
			functional	Checkall and Checknone cannot be used although a CD is set.
		During saving a track on the hard disk, the track played sounds jerkily.	sequencing	
		REW /rewind) farther than begin of a track does not start the track before.	functional	
		Record Shuffle does not cause shuffling the tracks; the track list is proceeded sequentially.	sequencing	
		In the Pause mode, pushing the Record button causes to play the track.	sequencing	
			functional	After activating Checkall and Checknone, the system doe not recognize that the action has been concluded, i.e., the related buttons are not enabled.
			functional	A song that is played during Record can be neither rewound nor fast forwarded.
		Checkall / Checkone++ and Checkoneoff cause during play jerking.	sequencing	
	3	Record Control and Eject causes removing the CD without warning.	functional	
		Activating Shuffle causes jerking the replay.	sequencing	

			sequencing	Multiple changes of songs recorded cause the warning that PC performance would not be sufficient a replay.
	4	Temporarily no jump to the selected track, and Stop of the replay although not all of the selected tracks are replayed.	sequencing	
3	2	In AutoPlaylist a new Playlist can be created. If desired, then should the way around be also possible, i.e., a new AutoPlaylist should be created out of a Playlist.	functional	
	3	Play replays the active playlist; Remix can only be activated at Stop.	functional	
4	2		sequencing	If neither a Genre nor an Artist is selected while creating a new AutoPlaylist, every track is listed in the appropriate list.
		Tacks that are recorded from a CD are temporarily not included In the actual list,	functional	
	3	If the menu entry File->Move is disabled, no track can be moved.	sequencing	
		Deleting the Track Info Styles starts the Windows Explorer also from which the tracks can be replayed.	functional	
6	2	NBCi Homepage cannot be visited because the link is obsolete.	functional	
			functional	Quick pushing of MySimon-ad (bottom right) after clicking any link triggers the IE error message „Page cannot be displayed“.

			functional	Unmotivated, random error message: "Audio Instant Message Error in Program Real-Jukebox"
7	2		functional	When Reset is pushed, the button should be disabled until any radio button has been touched.
			functional	If Vis.Settings are activated, all of the other windows should be blocked until close button is pressed.
			functional	If RJB is minimized, then also Vis. is minimized.
	3		sequencing	The list of the song titles of Vis. in Un-dock-Mode does not adjust if AllTracks of the view bar is not clicked at the moment of changing to another song.
			sequencing	If Vis.Settings are opened, prev and next Vis. toggle Slide features.
		When another task active, changing the size of the Vls. windows causes switching to RJB.	sequencing	
	4		sequencing	Changing the Vls. in Undoc mode does not change into Dock mode when closing the Undock mode.
			functional	Special Effects should be disabled if they do not control any effect.
		When Vls. window in Dock mode is clicked and moved several times during the	functional	

		window is settled, causes the settling to be abandoned.		
		A newly installed Vls. cannot be removed.	functional	
8	3		sequencing	An active Skin can be deleted.
		Clicking Delete Skin makes Explorer open.	functional	
	4	In Skin mode, clicking Play twice causes Stop, not Pause as expected.	functional	
			sequencing	In Skin mode, minimizing of the window and immediately maximizing it moves the window up to left, northwest.
			sequencing	Random error: Closing in Skin mode blocks an immediate starting right after.
11	2	After a search and replaying the tracks found, the original Playlist is forgotten.	sequencing	
			functional	Searchnow should disable the buttons Search and Matching until the end of the search process.
		Shuffle Mode does not function during a search process.	sequencing	
	3	Changing from SearchInternet to Searchalltrack via Menu never functions.	sequencing	
	4		sequencing	During replaying a track out of a track list, a search and the Stop of the track thereafter causes the track list of the search step to be active.
12	4		sequencing	A re-opening of Preferences, followed by moving the

				window and clicking from Display reveals a graphical defect for a short time.